Fault-Tolerant Reconfiguration of Trees and Rings in Distributed Systems *

Anish Arora

Computer Science
The Ohio State University
Columbus, OH 43210

Ashish Singhai

Computer Science
University of Illinois
Urbana-Champaign, IL 61801

Abstract

We design two programs that maintain the nodes of any distributed system in a rooted spanning tree and in a unidirectional ring, respectively, in the presence of any finite number of fail-stop failures and repairs of system nodes and communication channels. Our programs are fully distributed, have optimal time and space complexity, and illustrate two different methods for the design of nonmasking fault-tolerant programs.

Categories and Subject Descriptors

C.2.4	[Computer Communication Systems]	Distributed Systems
D.1.3	[Programming Techniques]	Concurrent Programming
D.2.4	[Program Verification]	Reliability
D.2.10	[Program Design]	Methodologies
D.4.5	[Operating Systems]	Fault-tolerance
G.2.2	[Discrete Mathematics]	Graph Algorithms

[0]Research supported in part by NSF grant CCR-9308640 and OSU Grant 221506

1 Introduction

Cooperation between the nodes of distributed systems is commonly realized by organizing the nodes into a convenient logical structure such as a ring, a star, or a tree. Such cooperation, however, poses a problem if faults can occur during the computation of the system: Fault occurrences can perturb the logical structure and thereby necessitate a reorganization of the nodes.

An ideal solution to the problem of reorganizing systems nodes into the desired logical structure is to design a program that is "nonmasking fault-tolerant". Nonmasking fault-tolerant programs make no assumptions about the predictability of fault occurrence and, hence, tolerate further perturbations to the structure that may occur while they reorganize the nodes. We call these programs nonmasking as they make no attempt to mask the perturbed structure from the cooperative mechanisms that rely on the structure.

Remarkably, methods for the design of nonmasking fault-tolerant programs have received little attention. This is largely due to the supposition that such methods will be complex, and will explicitly consider all combinations of the number, time, and duration of fault occurrences. Our experience shows, however, that this supposition is false [1-3]. We find that a few, simple methods suffice for the effective design of most nonmasking fault-tolerant programs, and avoid the combinatorial problem noted above. In this paper, we discuss two such methods for design of programs that reorganize system nodes.

One method is to compute the set X of all structures that result from perturbing the desired structure x by 0, 1, 2, ... simultaneous faults. Then, derive a 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 method follows from the fact that even if faults occur while the program is executing then, by (i), 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.

Another method is to make no assumption about the structure that results from the execution of faults. Instead, ensure that the program is "stabilizing" [4-5], in the following sense. Starting from an arbitrary program state, subsequent computation of the program (at the functional nodes) 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 the rest of this paper, we formulate the two methods described above in terms of a uniform definition of fault-tolerance [1-2]. Using these methods, we design two fully distributed, nonmasking fault-tolerant programs that maintain the nodes in an arbitrary but connected distributed system in a rooted spanning tree and in a unidirectional ring, respectively. More specifically, our programs tolerate any finite number of fail-stop failures and repairs of nodes and channels in the distributed system.

Our programs are optimal in their time and space complexity: Both reach a fixpoint within $O(N)$ time, where N is the number of system nodes that are "up" (i.e., currently not stopped due to a fail-stop failure). Our program for ring maintenance, infact, uses as a basis any nonmasking fault-tolerant program for maintaining a rooted spanning tree, and reaches a fixpoint within *constant* time of the latter reaching a fixpoint. Both programs use $O(log\ M)$ space at each node and in each message, where M is the total number of nodes.

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 nodes in the ring is at most 3. Both these "length" and "dilation" bounds are optimal since there exist networks that have no ring embedding of length less than $2(N-1)$ or dilation less than 3. (A simple example illustrating the lower bound on the length is a network where nodes are organized in a line [Figure 1a]; and for the lower bound on the dilation is a 3 by 3 mesh network with any two parallel lines deleted that are a unit distance apart [Figure 1b].)

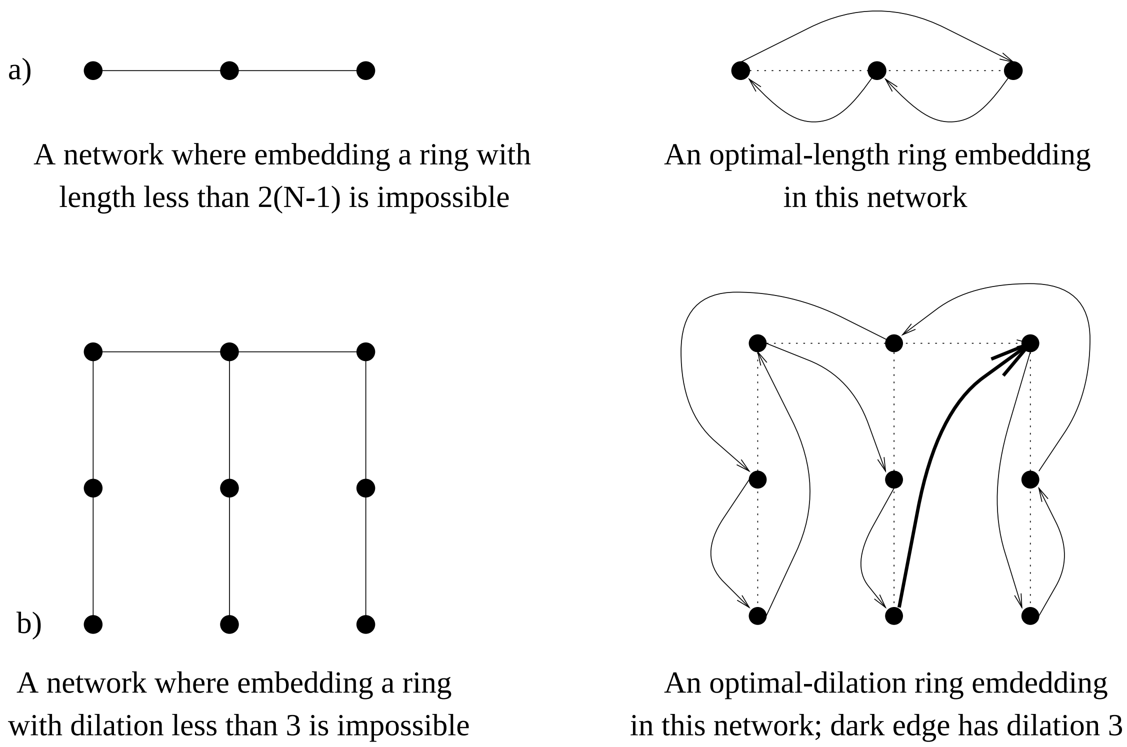


Figure 1: Lower bounds on length and dilation of ring embeddings

Our programs differ significantly from extant distributed algorithms for spanning tree construction and ring embeddings. More specifically, for the case of spanning tree construction, we recall that Gallagher et al [6] have presented an elegant solution but their solution is fault-intolerant, i.e., their solution does not solve the spanning tree reconfiguration problem. Stabilizing algorithms for spanning tree reconfiguration have been presented [7, 8, 9], that tolerate failures as well as repairs of both nodes and edges, but these programs are significantly more complex than our algorithm, since unlike our algorithm they allow for the formation of transient cycles in the graph of the parent variables. Moreover, these programs converge slower than our algorithm in the average case. Perlman's solution [9] additionally uses periodic transmission of messages, which results in significant communication overhead. Other programs we are aware of are at best tolerant only to the failures of edges or only the failures and repairs of nodes.

For the case of ring embeddings, the distributed solutions we are aware of [10, 11] do not tolerate fail-stop failures or repairs. Hélary and Raynal [10] have presented an algorithm that enumerates ring nodes, one at a time, in the order of a depth first traversal of a tree. While this limits the length of the embedded ring to $2(N-1)$, it allows the dilation to be linear in N. The message complexity of their algorithm is $O(N)$, as messages contain a list of nodes already traversed. By way of contrast, a message passing version of our algorithm would use messages of only $O(log\ M)$ size. Rosenstiehl et al [11] have presented a self-synchronizing tree network of finite state automata that execute in unison from a given initial state to embed a ring of dilation at most three on the tree edges. Again, the ring is formed one node at a time, unlike in our algorithm where a ring is embedded in a tree in constant time.

The paper is organized as follows. In Section 2, we state our assumptions and give a formal definition of programs and their nonmasking fault-tolerance. In Section 3, we use the first method 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

A distributed system consists of M nodes, that are named $1, ..., M$, and channels, that each connect a unique pair of nodes. At any instant, each node and channel is either "up" or "down". Only up nodes can execute actions to maintain the desired structure. Actions of an up node may involve communication only with up nodes that are "adjacent" to that node; i.e., connected by up channels. Channels are bidirectional. We henceforth use interchangeably the terms "node" and "up node" and the terms “channel” and “up channel”.

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

2.2 Programs

A “program” is a set of “variables” and a finite set of “actions”. Each variable has a predefined nonempty domain. Each action has the form:

$$\langle \text{guard} \rangle \quad \rightarrow \quad \langle \text{statement} \rangle$$

A guard is a boolean expression over the program variables. A statement updates zero or more program variables and always terminates upon execution.

Let p be a program. A “state” of p is defined by a value for each variable in each node 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.

Finally, a state predicate Q “converges to” R in p iff Q and R are closed in

p and, starting from any state where Q holds, every computation of p has a state where R holds.

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 state 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 reaches in the presence of faults can be characterized by a state predicate T that is closed in p. Such a state predicate identifies the "fault-span" of p. Note that the fault-span includes the fault-free states of the program. (Hence $S \Rightarrow T$.) Examples showing how to design fault-span predicates appear in [2]. These examples take the view that all classes of faults can be represented as actions that change the program state. For instance, a fail-stop failure of a node can be represented as an action that marks an up node as being down, and a repair can be represented as an action marks a down node as being up and reinitializes the state of that node.

We are now ready to give a uniform definition of fault-tolerance. Let S be the invariant and T be the fault-span of p (hence, S is a fixpoint of p and T is closed in p). And let F be a set of fault actions. For p to be $F-$tolerant, the following two requirements should be satisfied :

- Closure: T is preserved by the actions of F ,
- Convergence: T converges to S in p .

The definition above enables a formal classification of masking and nonmasking fault-tolerance [1-2]. Let p be F-tolerant. If $T = S$, we say that p is masking $F-$tolerant. Else ($T \neq S$), we say that p is nonmasking $F-$tolerant.

We observe finally that the first method discussed in Section 1 is formulated in terms of this definition by requiring that T implies that the structure is in the computed set X; and the second method T is formulated by requiring that T is the state predicate $true$.

3 Maintaining a Rooted Spanning Tree

In this section, we employ the first of the two design methods discussed in Section 1 to derive a nonmasking fault-tolerant program for maintaining a rooted spanning tree. Recall that in this method 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.

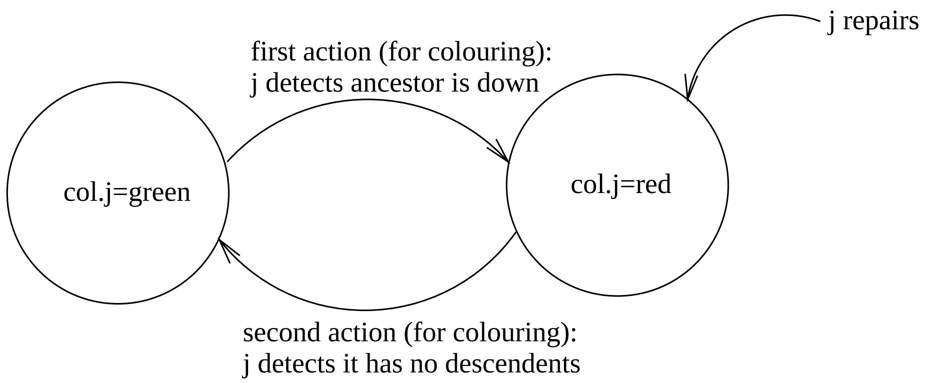
Let us represent the rooted spanning tree in terms of a "parent" relation between nodes; each node j maintains the index of its current parent node in the tree. Now, if any number of nodes or channels fail-stop, the resulting graph of the parent relation of the up nodes is a forest of trees. Moreover, if any number of nodes or channels repair, the resulting graph of the parent relation of the up nodes is also a forest, provided each node initializes itself to be its own parent when it repairs, and leaves its parent unchanged when a channel incident at it repairs.

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

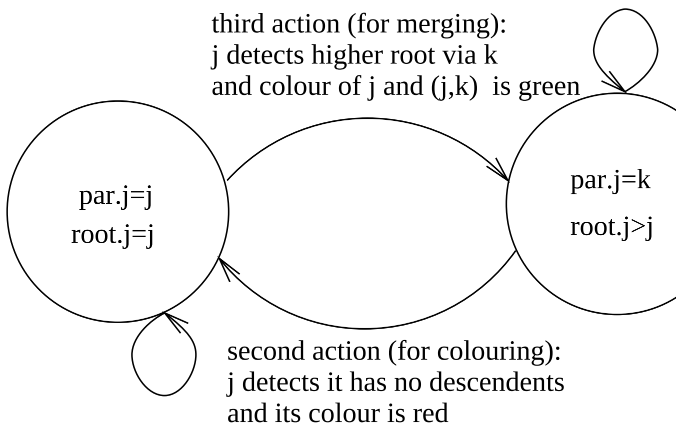
To derive our program, we need to address two problems. First, how to handle trees that are not rooted, that is, trees with nodes whose parent node or "parent channel" (i.e., channel to the parent node) are down. And second, how to handle multiple rooted trees in the graph of the parent relation of up nodes.

To solve the first problem, we need a mechanism to correct each node that has a down ancestor node or channel. We propose to correct such nodes, while maintaining the structure to be in X, by associating a colour with each node, as follows. As long as all ancestor nodes and channels of a node j are up, the colour of j is green. When a green coloured j detects that its parent node or channel is down or is coloured red, j colours itself red. Also, when a red coloured j detects that it has no remaining children, it disowns its parent and elects itself root; to do so, it makes itself its own parent and colours itself green. The net effect of these colouring actions is that when any node or channel goes down (i) all descendents of that node or channel colour themselves red and then (ii) all descendents of that node respectively disown their parent. In other words, eventually a state is reached where the color of a node is green if and only if the color of all of its ancestor nodes and channels is green.

Observe that a node or channel, whose fail-stop makes its children colour themselves red, may repair before the children recolour themselves green. In this case, we require that upon repair the node or channel in question colours itself red.



To solve the second 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 node j maintains a root value denoting the index of the node 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 node 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 nodes along all tree paths from their root to their leaves is always nonincreasing and (ii) eventually the root value of each node is the index of the root of its tree.



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

Our derivation above suffices to ensures that, starting from any structure in X, the node with the highest index will elect itself as a root and colour itself green via a colouring action, following which the remaining nodes will join its tree via merge actions.

A formal description of our program now follows. Node j maintains the parent, root, and colour values in variables $par.j$, $root.j$, and $d.i$, respectively. Let $Adj.j$ be the set of nodes adjacent to j. The node has three actions: The first action is used to colour j red; as described above, this action is enabled when

$$col.j = green \;\wedge\; (par.j \notin Adj.j \cup \{j\} \;\vee\; col.(par.j) = red \;\vee\; col.(j,(par.j)) = red)$$

The second action is used to disown the parent of j and to recolour j green; as described above, this action is enabled when

$$col.j = red \wedge \;\; (\text{for each node } k, k \in Adj.j, par.k \neq j)$$

The last action is used to merge j into the tree containing k; as described above, this action is enabled when

$$k \in Adj.j \;\wedge\; col.j = green \;\wedge\; col.(j,k) = green \;\wedge\; col.k = green \;\wedge\; root.j < root.k$$

Remark on notation : We use "parameters" to specify the third action. A parameter ranges over a finite domain. Its function is to enable a set of actions to be represented as one parameterized action. For example, let m be a parameter whose value is 0, 1 or 2; then the parameterized action $act.m$ abbreviates the following set of three actions.

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

(End of remark.)

Our program, RST, for maintaining a rooted spanning tree therefore consists of the following three actions for each node:

node $j \;\; (j : 1..M)$

var $root.j, par.j, : 1..M;$

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

parameter $k : 1..M;$

actions

$$
\begin{array}{lll}
col.j = green \;\wedge & & \\
(par.j \notin Adj.j \cup \{j\} \;\vee\; col.(par.j) = red & & \\
\qquad \vee\; col.(j, (par.j)) = red) & \longrightarrow & col.j := red \\
\rrbracket & & \\
col.j = red \;\wedge & & \\
(\forall k : k \notin Adj.j \;\vee\; par.k \neq j) & \longrightarrow & col.j, col.(j, par.j), root.j, par.j := green, green, j, j \\
\rrbracket & & \\
k \in Adj.j \;\wedge & & \\
col.j = green \;\wedge\; col.(j,k) = green \;\wedge & & \\
col.k = green \;\wedge\; root.j < root.k & \longrightarrow & root.j, par.j := root.k, k
\end{array}
$$

Program *RST*

We illustrate Program RST by an example. Figure 2(a) shows a 5-node system at an invariant state where node 4 is the root of the spanning tree, since node 5 is down. Consider, now, that node 4 and channel (1,2) fail. In this scenario, nodes 1 and 2 colour themselves red by their first actions, since their parent channel and parent node, respectively, are down. Subsequently, node 3 colours itself red by its first action when it detects that its parent, node 2, is coloured red (Figure 2(b)). Since nodes 1 and 3 have no children they can elect themselves as root by their second actions (Figure 2(c)), and since node 3 is the highest up node, nodes 1 and 2 will join its tree by their third actions. If node 5 now repairs, it will colour itself red (Figure 2(d)), and since it has no children it will eventually colour itself green by its second action, after which all nodes will join its tree by executing their third actions (Figure 2(e)).

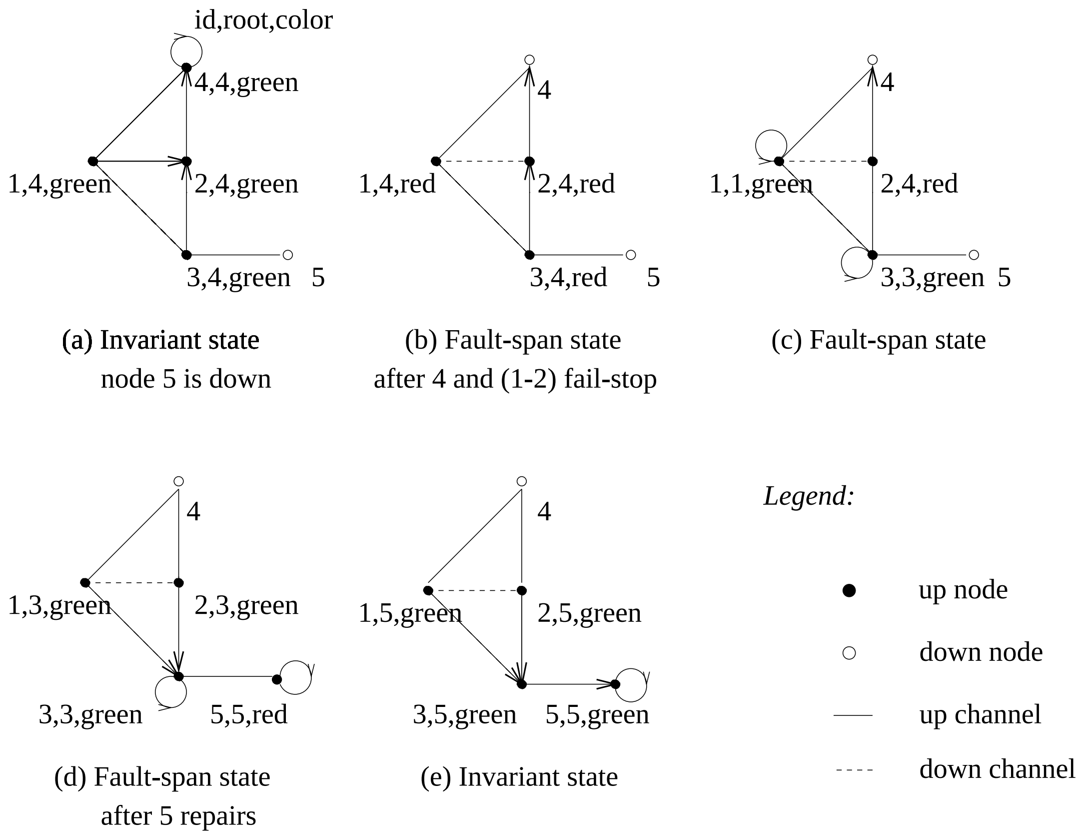


Figure 2

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 $T =$

$$
\begin{array}{ll}
& \text{graph of the parent relation is a forest of trees} \\
\wedge & (\forall j : \ j \text{ is up} \ \Rightarrow \\
& \quad\quad (col.j = red \quad \Rightarrow \quad (par.j \notin Adj.j \cup \{j\} \ \vee \ col.(par.j) = red \ \vee \ col.(j,(par.j)) = red)) \\
& \quad \wedge \ (par.j = j \quad \Rightarrow \quad root.j = j) \\
& \quad \wedge \ (par.j \neq j \quad \Rightarrow \quad root.j > j) \\
& \quad \wedge \ (par.j \in Adj.j \quad \Rightarrow \quad (root.j \leq root.(par.j) \ \vee \ col.(par.j) = red \ \vee \ col.(j,(par.j)) = red))) \\
\wedge & (\forall\ j,k : \ (j,k) \text{ is up} \ \wedge \ col.(j,k) = red \quad \Rightarrow \\
& \quad\quad (par.j = k \ \wedge \ (col.j = red \ \vee \ root.j > root.k)) \\
& \quad \vee \ (par.k = j \ \wedge \ (col.k = red \ \vee \ root.k > root.j)))
\end{array}
$$

And let $S =$

$$
\begin{array}{l}
\quad T \\
\wedge\ (\forall j:\ j \text{ is up} \ \Rightarrow \\
\qquad\qquad (col.j = red \ \Leftarrow\ (par.j \notin Adj.j \cup \{j\} \ \vee\ col.(par.j) = red \ \vee\ col.(j,(par.j)) = red)) \\
\qquad\quad \wedge\ col.j = green \\
\qquad\quad \wedge\ (\forall k: k \notin Adj.j \ \vee\ root.j = root.k))
\end{array}
$$

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 forest of trees. We show that this forest consists of exactly one tree: At each state where S holds, all nodes are coloured green, have a parent that is up, and have the same root value. Moreover, all channels are coloured green. It follows that there is exactly one node j satisfying $root.j = j \ \wedge\ par.j = j$ and, hence, exactly one tree. □

Lemma 2 *S is a fixpoint of RST.*

Lemma 3 *T is closed in RST.*

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

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

Proof (sketch): 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, $U \equiv (T \wedge (\forall j : j \text{ is up} \ \Rightarrow\ (col.j = red \ \Leftarrow\ (par.j \notin Adj.j \cup \{j\} \ \vee\ col.(par.j) = red \vee col.(j,(par.j)) = red))$. 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. □

We measure the time complexity of convergence to S in terms of rounds [7]. A round is a minimal, nonempty sequence of program steps wherein for each node there exists a step where the node 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 nodes. It follows that the time complexity of RST is $O(N)$ rounds.

We refer the reader to [12] for formal proofs of all lemmas and theorems in this paper.

4 Maintaining A Unidirectional Ring

In this section, we employ the second of the two design methods discussed in Section 1 to derive a nonmasking fault-tolerant program for maintaining a unidirectional ring. Recall that in this method, 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, is established.

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

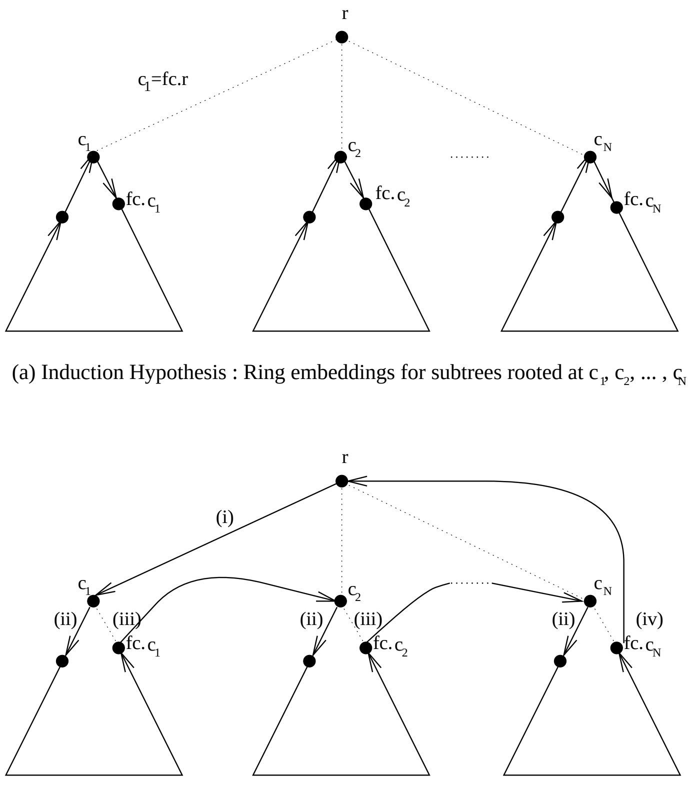
Lemma 6 *A unidirectional ring embedding of length at most* $2(N-1)$ *and dilation at most 3 can be embedded in an arbitrary but connected graph.*

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. In particular, 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.



(a) Induction Hypothesis : Ring embeddings for subtrees rooted at $c_1, c_2, \ldots, c_N$

(b) Induction Step : Ring embedding for tree rooted at r
Note the reversal of subtree embeddings

Figure 3

Consider the children of r in some fixed total order $c_1, c_2, ...c_N$, where $c_1 = fc.r$ (Figure 3a). (i) Connect r to c_1. (ii) Invert the successor relation of each $ring.c_j$, $1 \leq j < N$. (iii) Delete from each $ring.c_j$, $1 \leq j \leq N$, the edge from $fc.c_j$ to c_j, and connect $fc.c_j$ to c_{j+1}. (iv) Delete from $ring.c_N$ the edge from $fc.c_N$ to c_N, and connect $fc.c_N$ to r. The resulting structure (Figure 3b) is a ring that satisfies the properties 1-2. $\square$

In accordance with the construction 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.

We may implement the first layer in terms of any nonmasking fault-tolerant program for maintaining a rooted spanning tree, e.g., Program RST of Section 3. For our purposes, we merely assume that the following variables describing the rooted spanning tree are available to the second layer of each node j.

- $par.j$, denoting the parent node of j ($par.j = j$ if j is the root node).
- $fc.j$, denoting respectively the first node in some fixed total ordering of the children node 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 node 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 node. (To see that $even.j$ is implemented without adding to the time complexity of the first layer program, observe that its actions can be augmented to maintain a variable $even.j$. The even value of j is true, if j is a root node, and is the negation of the even value of $par.j$, if j is not a root node.)

We now describe the second layer. Let us represent the unidirectional ring in terms of a "successor" relation; the second layer in each node j maintains the index of the current successor 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 nodes j of the spanning tree, as follows. Since the successor 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 node j in one step, as follows:

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

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

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

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

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

---

node $j \ \ (j : 1 \,..\, M)$
owned var $par.j, fc.j, ns.j : 1..M$;
$even.j : Boolean$;
shared var $s.j : 1..M$;

actions

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

Program *UR*

We illustrate Program *UR* with the example of Figure 2. Let us order the children of nodes spatially from left to right. In Figure 2(a), the only child of the root node, 4, is node 2; hence $s.4 = 2$ by the first action. The children of node 2 are 1 and 3, node 1 is childless, and $\neg even.2$ holds; hence $s.1 = 5$ by the second action. Node 3 satisfies $even.3$ and has no sibling to the right; hence, $s.2 = 3$ by the third action. Finally, node 1 is childless and node 3 is its next sibling; hence $s.3 = 1$ by the fifth action. Thus, Program UR yields the unidirectional ring $(4, 2, 3, 1)$. Likewise, in Figure 2(e), $s.5 = 3, s.3 = 2, s.2 = 1$, and $s.1 = 5$ and, thus, Program UR yields the unidirectional ring $(5, 3, 2, 1)$.

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 node j, there is some node k (possibly identical to j) such that* $s.j = k$.

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

Lemmas 7 and 8 imply that at each state where S holds the in-degree and out-degree of each node j in the graph of the successor relation is 1. Hence, the graph of the successor relation is a collection of node 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 h that all nodes of a subtree of height h belong to the same ring. Let the fact that two nodes 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 subtree consists of a single node, hence the hypothesis is trivially satisfied.

Induction Step $(h > 0)$: From the induction hypothesis, if nodes k and l belong to the subtree rooted at a child c of r, then $k =_r l$. Since action 1 of node 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 nodes 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 fixpoint state of the first layer, every computation of UR reaches a state where S holds within 1 round.* □

5 Concluding Remarks

In this paper, we demonstrated two efficient, nonmasking fault-tolerant algorithms for reorganizing system nodes into trees and rings respectively. The algorithms were designed effectively using two formal methods and were noted to possess several features that are lacking in extant distributed algorithms for spanning tree construction and ring embedding.

The first formal method was based on computing the system structures resulting from any number of simultaneous fault occurrences; this method was appropriate when such computation was easy, as was the case in maintaining trees where the perturbed system structure was readily shown to be a forest of trees. The second formal method made no assumptions about the system structure resulting from fault occurrences; this method was appropriate when computation of the system structure in the presence of faults was either hard or yielded a complex structure that was inappropriate for design purposes, as was the case in maintaining rings.

We recall that a standard approach to reconfiguration in distributed systems is to elect one node as the leader, that will control the process of configuration. Leader election is readily achieved by electing the root of the rooted spanning tree as the leader. Our algorithm RST can, therefore, be also viewed as a nonmasking fault-tolerant solution to the leader election problem.

We also note that our proofs of convergence do not depend on the assumption of fairness for execution of continuously enabled actions. We have thus far assumed fairness only to simplify the exposition. Moreover, there exist refinements of RST and UR that yield read/write atomicity algorithms that are nonmasking fault-tolerant as well. We conjecture that the resulting read/write atomicity programs can be nicely translated into nonmasking fault-tolerant message passing programs.

While we have focused our attention on the design of nonmasking fault-tolerant reconfiguration algorithms in this paper, it will be worthwhile to extend our nonmasking fault-tolerant algorithms to obtain masking fault-tolerant algorithms; that is, algorithms that mask the perturbed structure from the cooperative mechanisms that rely on the structure.

In some cases, we anticipate the result of extending nonmasking algorithms will be algorithms that are both masking and nonmasking fault-tolerant. The design of such algorithms is motivated by the insight that the fault-span of an algorithm (namely T) need not be unique [1-2]. It follows multiple fault-spans may be associated with an algorithm. Hence, it is possible that an algorithm is nonmasking tolerant with respect to one of these fault-spans and nonmasking tolerant with respect to another.

Other extensions will include extending the nonmasking fault-tolerant algorithms so that they are secure during the period of convergence. Security is especially important in modern systems where reconfiguration occurs frequently and thus the system is often in vulnerable states.

Acknowledgements. We thank the anonymous referees for their constructive suggestions.

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

[3] A. Arora, M. G. Gouda, and G. Varghese, "Constraint satisfaction as a basis for designing nonmasking fault-tolerance", *Journal of High Speed Networks*, (1994 to appear); *Proceedings of the 14th International Conference on Distributed Computer Systems* (1994), pp. 424–431.

[4] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control", *Communications of the ACM* , Vol. 17, No. 11, pp. 643-644 (1974).

[5] 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.

[6] 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.

[7] A. Arora and M. G. Gouda, "Distributed reset", *IEEE Transactions on Computers* 43(9) (1994).

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

[9] 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.

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

[11] 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.

[12] A. Arora and A. Singhai, "Optimal, nonmasking fault-tolerant reconfiguration of trees and rings", *OSU Technical Report CISRC-TR09-1994*; a preliminary version appears in *Proceedings of International Conference on Network Protocols*, pp. 221-228, 1994.