Efficient Reconfiguration of Trees: A Case Study in Methodical Design of Nonmasking Fault-Tolerant Programs *

.5em Anish Arora

Department of Computer Science
The Ohio State University
Columbus, OH, USA 43210
anish@cis.ohio-state.edu

=1cm

Abstract. We illustrate a formal method for the design of nonmasking fault-tolerant programs, by demonstrating how the method enables us to effectively design a new and efficient program. Our program maintains the processes of any given distributed system in a spanning tree, tolerates any finite number of fail-stop failures and repairs of system processes and channels, and requires only $O(n)$ time and $O(n\ log\ n)$ space to reconfigure the tree, where n is the number of nonfaulty processes. The program is, moreover, simple and fully distributed.

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
G.2.2	[Discrete Mathematics]	Graph Algorithms

* Research supported in part by NSF Grant CCR-9308640 and OSU Grant 221506.
A preliminary version of this paper appears in the Proceedings of the Third International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems, 1994

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 network nodes into the desired logical structure is to design a protocol that is "nonmasking fault-tolerant". Nonmasking fault-tolerant protocols 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 protocols 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 protocols 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 faults occurrences. Our experience shows, however, that this supposition is false. We find [1-3] that a few, simple methods suffice for the effective design of most nonmasking fault-tolerant protocols, and avoid the combinatorial problem noted above. Below, we discuss two such methods for design of protocols to reorganize network 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 protocol so that starting from any protocol state where the structure is in X, subsequent computation of the protocol (i) always yields protocol states where the structure is in X and (ii) reaches within a bounded number of steps a protocol state where the structure is x. The correctness of this method follows from the fact that even if faults occur while the protocol is executing then, by (i), the protocol is always in a state where the structure is in X; hence, subsequent computation of the protocol 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 protocol is "stabilizing", in the following sense. Starting from an arbitrary protocol state, subsequent computation of the protocol converges within a bounded number of steps to a protocol state where the structure is x. Correct execution of the computer network then resumes, and continues until a subsequent fault occurrence perturbs the structure, in which case the cycle of convergence repeats.

In this paper, we focus our attention on the design of programs that are "nonmasking" fault-tolerant. Informally, a program is nonmasking fault-tolerant if faults violate the input-output relation of the program only temporarily. Typically, such a program tolerates faults by restoring its state, by replaying its computation, or by repairing itself, whenever it detects a violation of its input-output relation.

At a first glance, it might appear that it is ideal to always design programs that are "masking" fault-tolerant, whereby faults never violate the input-output relation of the program. It turns out, however, that there are several situations where designing masking fault-tolerance is impossible, impractical, or unnecessary. The study of alternative design methods is therefore desirable.

Remarkably, methods for the design of nonmasking fault-tolerance have received little attention thus far. Most of the alternative design methods in use today yield only partial fault-tolerance, for instance,

- "Offline recovery", wherein the program executes "recovery" actions after each fault occurs or after all faults stop occurring. Once the recovery actions have completed execution, the program restarts its "normal" actions. The recovery actions do not themselves tolerate the faults; they, therefore, execute "offline".

- "Welldefined failure", wherein the program executes recovery actions in the presence of fault occurrences. The recovery actions attempt to mask all faults occurrences, but when they cannot do so they "fail" the program in a "welldefined" manner, for example, by halting the program or by aborting the progress of some process.

  The program may also execute normal actions in the presence of fault actions. When the execution of recovery actions interferes with that of the normal actions being executed concurrently, the recovery actions, again, fail the program in a welldefined manner.

Why have design methods for nonmasking fault-tolerance received little attention? We conjecture that this is due to the following extant beliefs: (1) It is hard to design recovery actions that execute correctly in the presence of multiple faults, as every combination of the number, time, and duration of faults occurrences has to be considered. (2) It is easier to ensure that the program fails in a welldefined manner than to ensure that the program eventually reestablishes its input-output relation. (3) It is hard to design recovery and normal actions of the program that do not interfere with each other when executed concurrently.

Our experience suggests that these beliefs are invalid in many cases. We find that a method based on a formal definition of fault-tolerance [1] and a few structuring principles [1-2] suffices for the design of many nonmasking fault-tolerant programs. Examples that we have already designed using this method include programs for election, routing, distributed reset, mutual exclusion, clock unison, diffusing computations, termination detection, ring formation, global state detection and monitoring, atomic actions and load balancing.

In this paper, we illustrate the method by demonstrating how it enables us to effectively design a new and efficient program. More specifically, our program maintains the processes of an arbitrary distributed system in a rooted spanning tree, and tolerates any finite number of fail-stop failures and repairs of system processes and communication channels. The program is fully distributed and requires only $O(n)$ time and $O(n\ log(n))$ space to reconfigure the tree, where n is the number of nonfaulty processes.

We proceeds as follows. In Section 2, we formalize the notions of programs, faults, and fault-tolerance, and also introduce our programming notation. In Section 3, we outline our design method. In Section 4, we present our case study in the use of the design method; to begin with, we restrict our design to tolerate fail-stop failures and repairs of system processes only; later in the section, we extend our design to tolerate fail-stop failures and repairs of communication channels as well. Finally, in Section 5, we make comparisons to related work and concluding remarks.

2 Programs, Faults, and Fault-Tolerance

2.1 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.

State. Let p be a program. A state of p is defined by a value for each variable of p, chosen from the domain of the variable. A state predicate of p is a boolean expression over the variables of p.

Closure. An action of p establishes a state predicate R iff upon starting from an arbitrary state of p, executing the statement of the action yields a state where R holds. An action of p preserves R iff upon starting from any state of p where R holds and the guard of the action is true, executing the statement of the action yields a state where R holds. R is closed in p iff each action of p preserves R.

Computation. A computation of p is a fair, maximal sequence of steps; in every step, the statement of some action in p whose guard is true in the current state is executed. Fairness of the sequence means that each action in p whose guard is continuously true along the states in the sequence is eventually executed. Maximality of the sequence means that if the sequence is finite then the guard of each action in p is false in the final state.

Invariant. One state predicate of p is distinguished as its invariant. Intuitively, the invariant of p is a predicate that characterizes the set of fault-free states of p, i.e., the states that are reached in the fault-free execution of p. Hence, every computation of p that starts at a state where the invariant of p holds is an intended computation, one that meets the (safety and progress properties of the) problem specification that p satisfies.

Since the invariant of p holds at every state in all computations of p that start at a state where the invariant holds, it follows that the invariant predicate is closed in p. Of course, many other state predicates may be closed in p, some of which need not be true at states where the invariant holds; the state predicate *false* provide a trivial example of such a predicate.

Convergence. 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. Note that the converges-to relation is transitive.

2.2 Faults

The faults that a program is subject to can be systematically represented by actions whose execution perturbs the program state. We emphasize that such representation is possible notwithstanding the type of the faults —be they stuck-at, crash, fail-stop, omission, timing, performance, or byzantine— or the nature of the faults —be they permanent, transient, or intermittent.

Fault-span. Consider a set of fault actions F that a program p is subject to. If actions in F execute while p is executing, the resulting states of p may no longer satisfy the invariant of p. Still, just as we can characterize the set of states that p reaches during fault-free execution, we can likewise characterize the set of states that p reaches when it executes in the presence of actions in F. We call this latter characterization the fault-span predicate of p with respect to F.

As may be expected, a fault-span predicate of p holds at each fault-free state of p. Hence, a fault-span predicate characterizes a set of states that includes the set of states characterized by the invariant. Like the invariant, a fault-span of p is, by definition, closed in p.

2.3 Fault-Tolerance

We are now ready to give a formal definition of fault-tolerance [1].

Let p be a program, F be a set of fault actions, and S be the invariant of p. We say that "p is F-tolerant for S" iff there exists a state predicate T of p such that the following three conditions hold:

- Inclusion: $T \Leftarrow S$
- Closure: T is closed in p and is preserved by each action in F
- Convergence: T converges to S in p

This definition can be understood as follows. At any state where the invariant, S, holds, executing an action in p yields a state where S continues to hold; but executing an action in F may yield a state where S does not hold. Nonetheless, the following three facts are true about this last state : (i) T, the fault-span, holds, (ii) subsequent execution of actions in p and F yields states where T holds, and (iii) when actions in F stop executing, subsequent execution of actions in p alone eventually yields a state where S holds, from which point the program resumes its intended execution.

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

3 Method for Designing Nonmasking Fault-Tolerance

In this section, we outline a method for the design of nonmasking fault-tolerant programs. The method is suggested by the definition given above for nonmasking fault-tolerant programs.

In the method, to design a program that satisfies a given problem specification and is nonmasking tolerant to a given set of fault actions, the following four design steps are performed:

1. *Design of the fault-span T*
   – A state predicate T is constructed that is weak enough so that the fault actions preserve it.
2. *Design of the invariant S*
   – A state predicate S is constructed that is strong enough to meet the safety properties of the problem specification.
3. *Design of program actions that achieve nonmasking tolerance*
   – Program actions are constructed that ensure T converges to S. (Recall that T converges to S implies that each of these actions preserves T as well as S.)
4. *Design of program actions that satisfy the problem specification*
   – Program actions are constructed to satisfy the problem specification

in all computations that start from states where S holds. Each of these actions is verified to preserve S as well as T.

The following general remarks elaborate upon the four design steps:

- The order in which the four steps are performed may vary from program to program. For example, in some cases it may be convenient to first design T and to then design S; the converse may be true in other cases.

- A step may have to be revisited if its consistency is violated by another step. For example, in Step 4 an action may be constructed that does not preserve T. As a result of this, Step 1 may have to be revisited, to weaken T. In turn, this may necessitate Step 3 to be revisited, to construct new actions that ensure the new T converges to S.

- Note that Step 1 is the only step that involves considering the fault actions.

- While Step 1 is required to yield a T that is weak nough to be preserved by the fault actions, it should not yield a T that is so weak as to complicate the design of Step 3 actions. Likewise, while Step 2 should yield an S that is strong enough to meet the safety properties of the problem specification, it should not yield an S so strong as to complicate the design of Step 4 actions.

- A convenient approach to designing the Step 3 actions is to identify the "constraints" in S. The constraints in S are state predicates that (i) can each be independently checked and established by some program action, and (ii) equivale S when taken in conjunction with each other and together with T. From (i), it follows that, for each constraint c in S, we can design one action of the form:

$$\neg c \quad \rightarrow \quad \text{“ establish } c \text{ and preserve } T\text{ ”}$$

in other words, an action that independently checks c and, if need be, establishes c while preserving T. From (ii), it follows that if every constraint is satisfied in a state where T holds, then S holds in that state.

In this approach, Step 3 actions preserve T by design. Also, since Step 3 actions execute only when $\neg S$ holds, they trivially preserve S as well and, moreover, do not interfere with the Step 4 actions in states where S holds.

Now, if Step 4 actions are constructed so as to preserve each constraint in S, it may follow that Step 4 actions do not interfere with Step 3 actions action in states where $T \wedge \neg S$ holds.

- The standard approach to verifying that Step 3 actions ensure T converges to S is to exhibit a "variant" function. A variant function is a mapping, from the set of program states to a set wellfounded under a relation $<$, satisfying the following condition: Upon starting from any state where $T \wedge \neg S$ holds, every computation step either decreases the value of variant function with respect to $<$ or yields a state where S holds.

Several other sufficient conditions for verifying convergence have been suggested in the literature. Some of these are specifically for constraint-based design [2]. For our case study, we will recall only the simple

Theorem: Let Q and R be closed state predicates of p.

> If p has an action whose guard is true if $Q \wedge \neg R$ holds and that establishes R, then Q converges to R in p. □

- Standard approaches to designing Step 4 actions appear in [3, 4].

4 The Case Study: Reconfiguration of Spanning Trees

In this section, we employ the method discussed above to design a nonmasking fault-tolerant program that satisfies the problem specification for reconfiguration of spanning trees.

4.1 Problem Specification

Given is an undirected, connected graph that consists of M nodes named $1, ..., M$. At each instant, each node in the graph is either "up" or "down". Two nodes in the graph are "adjacent" iff they are both are up and there is an edge between them.

Required is to design for each node j a set of actions that maintain a variable $p.j$, for "parent" of j, such that

-- actions of j may involve communication only with nodes that are adjacent to j,

-- the graph of the p variables of the up nodes is a rooted tree that spans all up nodes (and the root node of the tree is its own parent).

4.2 Fault Actions

Fault actions fail-stop and / or repair nodes and, thus, change the set of up nodes. (Fault actions that also fail-stop and / or repair edges and, thus, change the adjacency relation between nodes, are considered later in this section.)

For sake of simplicity, we assume that fault actions do not disconnect the set of up nodes; else, if the set is disconnected, the up nodes in each partition will reconfigure themselves into a separate rooted tree. We also assume that repair actions can reinitialize the state of the corresponding nodes or edges.

4.3 Designing the Fault-Span

In this subsection, in accordance with Step 1 of the method discussed in Section 3, we design the fault-span predicate T that is to be preserved by fault actions as well as program actions.

We begin by observing that if any number of nodes fail-stop starting from a state where the graph of the parent variables of the up nodes is a rooted spanning tree, then the resulting graph is a forest. Likewise, if any number of nodes repair then, assuming that each node j sets $p.j$ to j upon repair, the resulting graph is still a forest. We therefore postulate that T $\Rightarrow$ U,

$$U \;=\; graph\ of\ the\ parent\ variables\ of\ the\ up\ nodes\ is\ a\ forest$$

We next observe from U that the following two scenarios will arise in our program design. First, how to handle trees that are not rooted, i.e. trees that have a node whose parent is down. And second, how to handle multiple rooted trees.

To handle the first scenario, we propose to correct a node j that has an ancestor that is down, by introducing a variable $col.j$, for "color" of j. The value of $col.j$ is maintained by j to be $green$ as long as all ancestors of j are up; else, the value of $col.j$ is maintained by j to be red. In other words, j colors itself red iff its parent is down or is colored red. We can therefore postulate that T $\Rightarrow$ $(\forall j : j\ is\ up : \ T1.j)$,

$$T1.j \;=\; (col.j = red \;\Rightarrow\; (p.j \notin Adj.j \cup \{j\} \;\vee\; col.(p.j) = red))$$

where $Adj.j$ is the set of nodes adjacent to j. Observe that $(\forall j : j \ is \ up : T1.j)$ is preserved when nodes fail-stop. Likewise, $(\forall j : j \ is \ up : T1.j)$ is preserved when nodes repair, provided each j sets $col.j$ to red upon it repair. (Henceforth, all quantifications over j will be implicitly restricted to j such that $j \ is \ up$.)

To handle the second scenario, we propose to merge trees by introducing a variable $root.j$, for "root" of j . The value of $root.j$ is maintained by j to be the index of the root node of the tree that j is in. We can therefore postulate that $T \Rightarrow (\forall j :: T2.j)$,

$$T2.j \ = \ (p.j = j \ \Rightarrow \ root.j = j)$$

We propose that j merges with the tree to which an adjacent node k belongs iff $root.j < root.k$ holds, by setting $p.j$ to k and $root.j$ to $root.k$. We can therefore postulate that $T \Rightarrow (\forall j :: T3.j \wedge T4.j)$,

$$T3.j \ = \ (p.j \neq j \ \Rightarrow \ root.j > j)$$

$$T4.j \ = \ (p.j \in Adj.j \ \Rightarrow \ root.j \leq root.(p.j))$$

Observe that the state predicates $(\forall j :: T2.j)$, $(\forall j :: T3.j)$, and $(\forall j :: T4.j)$ are all preserved when nodes fail-stop. Likewise, $(\forall j :: T2.j)$ and $(\forall j :: T3.j)$ are preserved when nodes repair, provided j also sets $root.j$ to j upon its repair (recall that j sets $p.j$ to j upon its repair). Unfortunately, $(\forall j :: T4.j)$ may not be preserved when j repairs, since j sets $root.j$ to j which may be less than the $root$ variable of a child of j. Recalling that j also sets $col.j$ to red and $p.j$ to j upon repair, we weaken our definition of $T4.j$,

$$T4.j \ = \ (p.j \in Adj.j \ \Rightarrow \ (root.j \leq root.(p.j) \ \vee \ col.(p.j) = red))$$

and observe that the new $(\forall j :: T4.j)$ is preserved when nodes fail-stop and repair.

Now that we have decided how to handle both scenarios, our design of the fault-span predicate T is complete:

$$T \ = \ U \wedge (\forall j :: T1.j \wedge T2.j \wedge T3.j \wedge T4.j)$$

4.4 Designing the Repair Actions

We have assumed that when the repair "fault" occurs, each process j can reinitialize its state. Based on the decisions taken while designing T, upon repair j executes:

$$p.j, col.j, root.j := j, red, j$$

4.5 Designing the Program Invariant

In this subsection, in accordance with Step 2 of the method discussed in Section 3, we design an invariant predicate S that (i) satisfies the problem requirement and (ii) satisfies the inclusion requirement, $T \Leftarrow S$.

To handle the first problem requirement that each tree be rooted at an up node, we decided in Section 4.2 that each j be colored red if it has an ancestor that is down. Hence, we require that $S \;\Rightarrow\; (\forall j :: \; S1.j)$,

$$S1.j \;=\; (col.j = red \;\;\Leftarrow\;\; (p.j \notin Adj.j \cup \{j\} \;\vee\; col.(p.j) = red))$$

Moreover, to ensure that no node j has a down ancestor, we require that $S \;\Rightarrow\; (\forall j :: \; S2.j)$,

$$S2.j \;=\; col.j = green$$

To handle the second problem requirement that there exist only one tree that spans the up nodes, we decided in Section 4.2 that each j change its tree if it has an adjacent node k such that $root.j < root.k$. Hence, recalling that the adjacency relation is symmetric, we require that $S \;\Rightarrow\; (\forall j :: \; S3.j)$,

$$S3.j \;=\; (\forall k : k \in Adj.j \;\Rightarrow\;\; root.j = root.k)$$

Observe that at each state where $(\forall j :: \; S1.j \wedge S2.j \wedge S3.j)$ holds, the graph of the parent variables of the up nodes is a rooted tree that spans all up nodes. Hence, our design of the invariant predicate S is complete:

$$S \;=\; T \wedge (\forall j :: \; S1.j \;\wedge\; S2.j \;\wedge\; S3.j)$$

4.6 Designing the Program Actions

In this subsection, in accordance with Step 2 of the method discussed in Section 3, we design program actions that ensure T converges to S.

Based on the design decision taken in the previous subsection and bearing in mind the Theorem in Section 3, we propose to design for each node j three actions such that:

-- Action 1 establishes $S1.j$ and preserves T,

-- Action 2 establishes $S2.j$ and preserves T and $S1.k$ for each node k,

-- Action 3 establishes $S3.j$ and preserves T, $S1.k$ and $S2.k$ for each node k.

For Action 1, we consider:

$$\neg S1.j \quad \longrightarrow \quad col.j := red$$

and observe that this action establishes $S1.j$ and preserves T.

For Action 2, we begin by considering:

$$\neg S2.j \quad \longrightarrow \quad col.j := green$$

This action may, however, violate $T1.k$ and $T4.k$ if k is a red colored child of j. It also violates $S1.k$ if k is a red colored parent of j. We, therefore, refine the action:

$$\neg S2.j \ \wedge \ (\forall k : k \in Adj.j \ \Rightarrow \ p.k \neq j) \quad \longrightarrow \quad col.j, p.j := green, j$$

The refined action may, however, violate $T2.j$ if $root.j$ exceeds j. We, therefore, further refine the action:

$$\neg S2.j \ \wedge \ (\forall k : k \in Adj.j \ \Rightarrow \ p.k \neq j) \quad \longrightarrow \quad col.j, p.j, root.j := green, j, j$$

and observe that this action establishes $S2.j$ and preserves T and $S1.k$ for each k.

For Action 3, we begin by considering:

$$\neg S3.j \quad \longrightarrow \quad p.j, root.j := k, root.k$$

This action may, however, violate $T1.j$ if $col.j = red$, and $S1.k$ if $col.k = red$. We, therefore, refine the action:

$$\neg S3.j \ \wedge \ col.k = green \ \wedge \ col.j = green \quad \longrightarrow \quad p.j, root.j := k, root.k$$

and observe that this action establishes $S3.j$ and preserves $T, S1.k, S2.k$ for each k.

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

$$
\begin{array}{ll}
\textbf{node} & j \;\; (j : 1..M) \\
\textbf{var} & root.j, p.j : 1..M; \\
 & col.j : \{green, red\}; \\
\textbf{parameter} & k : 1..M; \\
\textbf{actions} &
\end{array}
$$

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

4.7 Verifying Convergence

Based on the transitivity of the converges-to relation and in keeping with the order in which we introduced actions in Section 4.6, our proof that T converges to S in program RST consists of 3 stages:

1. T	converges to	$T \wedge (\forall j :: S1.j)$	in RST
2. $T \wedge (\forall j :: S1.j)$	converges to	$T \wedge (\forall j :: S1.j \wedge S2.j)$	in RST
3. $T \wedge (\forall j :: S1.j \wedge S2.j)$	converges to	S	in RST

Proof of Stage 1: We use the fact that $S1.j$ is preserved by the actions of all nodes that are not ancestors of j in the graph of the p variables, as follows. By induction on the depth, d, of up nodes in their tree, we claim: T converges to $U.d$ in RST, where $U.d \;=\; T \wedge (\forall j : \text{depth of } j < d \;:\; S1.j)$. Observe that $U.d$ is closed in RST.

The base case, $d = 0$, is trivially true. For the induction case, $d > 0$, we observe that Action 1 of each up node k whose depth is $\leq d$ preserves $S1.j$ for each j whose depth is d. Also, Actions 2 and 3 of each up node k preserve $S1.j$. The claim now follows from the hypothesis that T converges to $U.(d-1)$, the fact that $U.(d-1)$ and $U.d$ are closed, the fact that $S1.j$ is preserved by the actions of all nodes that are not ancestors of j in the parent relation, the Theorem in Section 3, and the transitivity of the converges-to relation.

Proof of Stage 2: Consider the variant function: number of up nodes that are colored red. (We observe in this case that the variant function does not increase in any step and decreases eventually due to the fairness of action execution.)

Proof of Stage 3: Consider the variant function: sum of $M - root.j$ for all up j.

4.8 Adding Tolerance to Fail-stop Faults and Repairs of Edges

We conclude our case study by illustrating how to extend our design so that the resulting program tolerates fail-stop faults and repairs of edges, in addition to tolerating those of nodes.

We revisit, first, the design of the fault-span. As designed above, the only references to edges in the fault-span T are in $T1.j$ and $T4.j$ (both in terms of $p.j \in Adj.j$).

Note that $T1.j$ and $T4.j$ are preserved when edges fail-stop, but are violated whenever the edge between j and $p.j$ repairs in a state where $col.j = red \;\vee\; root.j > root.(p.j)$. Therefore, we need to weaken $T1.j$ and $T4.j$. To do so, we introduce a variable $col.(j,k)$, for "color" of the edge between j and k, that is set to red only if $T1.j$ and $T4.j$ may be violated due to an edge repair. We can now postulate that $T \Rightarrow (\forall\, j,k : (j,k)\ is\ up : \ col.(j,k) = red \;\Rightarrow\; R.(j,k))$,

$$R.(j,k) \;=\; ((p.j = k \;\wedge\; (col.j = red \vee root.j > root.k)) \;\vee \\ (p.k = j \;\wedge\; (col.k = red \vee root.k > root.j)))$$

and the weakened $T1.j$ and $T4.j$ are, respectively, :

$$col.j = red \;\Rightarrow\; (p.j \notin Adj.j \cup \{j\} \;\vee\; col.(p.j) = red \;\vee\; col.(j,(p.j)) = red)$$

and

$$p.j \in Adj.j \;\Rightarrow\; (root.j \leq root.(p.j) \;\vee\; col.(p.j) = red \;\vee\; col.(j,(p.j)) = red)$$

We observe that $(\forall j :: T1.j)$ and $(\forall j :: T4.j)$ are trivially preserved when edges fail-stop. Likewise, $(\forall j :: T1.j)$ and $(\forall j :: T4.j)$ are trivially preserved an edge between nodes j and k repairs, provided the predicate

$R.(j,k) \;\Rightarrow\; col.(j,k) = red$ holds. Our design of the new fault-span predicate T thus yields:

$$\begin{aligned} T \;=\;& U \wedge (\forall j :: T1.j \wedge T2.j \wedge T3.j \wedge T4.j) \\ & \wedge (\forall\ (j,k) ::\ col.(j,k) = red \;\Rightarrow\; R.(j,k)) \end{aligned}$$

We revisit, next, the design of repair actions. As noted above, we need to ensure that $R.(j,k) \;\Rightarrow\; col.(j,k) = red$ holds when the edge between j and k repairs. To do so, we design the following repair action for the edge between j and k:

$$\text{if } \ R.(j,k) \ \text{ then } \ col.(j,k) := red$$

We revisit, now, the design of the invariant. $S1.j$ had been designed based on the shape of $T1.j$. Due to the weakening of $T1.j$, we need to strengthen $S1.j$, correspondingly, to:

$$col.j = red \;\Leftarrow\; (p.j \notin Adj.j \cup \{j\} \ \vee\ col.(p.j) = red \ \vee\ col.(j,(p.j)) = red)$$

thus obtaining a new S that satisfies the problem requirement:

$$S \;=\; T \wedge (\forall j ::\ S1.j \wedge S2.j \wedge S3.j)$$

We revisit, finally, the design of the program actions. We need to modify Action 1 to preserve the new T (in particular, $T1.j$) and to establish the new $S1.j$; we need to modify Action 2 to preserve the new T (in particular $(\forall\ j,k : (j,k)\ is\ up :\ col.(j,k) = red \;\Rightarrow\; R.(j,k)))$; and we need to modify Action 3 to preserve the new $S1.j$. The modifications are straightforward, and we encourage the reader to attempt them. Below, we give a summary of the resulting program that tolerates any finite number of fail-stop failures and repairs of nodes and edges.

Summary of Program Design

Fault Span

$$
\begin{array}{lcl}
T & = & \textit{graph of the parent variables of up nodes is a forest} \\
 & & \wedge\ (\forall j : j \textit{ is up} : T1.j\ \wedge\ T2.j\ \wedge\ T3.j\ \wedge\ T4.j), \\
 & & \wedge\ (\forall\ j,k : (j,k) \textit{ is up} : col.(j,k) = red\ \Rightarrow\ R.(j,k)), \quad \text{where} \\
T1.j & = & (col.j = red \quad \Rightarrow \quad (p.j \notin Adj.j \cup \{j\}\ \vee\ col.(p.j) = red\ \vee\ col.(j,(p.j)) = red)) \\
T2.j & = & (p.j = j \quad \Rightarrow \quad root.j = j) \\
T3.j & = & (p.j \neq j \quad \Rightarrow \quad root.j > j) \\
T4.j & = & (p.j \in Adj.j \quad \Rightarrow \quad (root.j \leq root.(p.j)\ \vee\ col.(p.j) = red\ \vee\ col.(j,(p.j)) = red)) \\
R.(j,k) & = & ((p.j = k\ \wedge\ (col.j = red\ \vee\ root.j > root.k))\ \vee \\
 & & (p.k = j\ \wedge\ (col.k = red\ \vee\ root.k > root.j))))
\end{array}
$$

Repair Actions

Repair of j: $\quad j \textit{ is down} \quad \longrightarrow \quad j \textit{ is up}\ ,\ p.j, col.j, root.j := j, red, j$

Repair of (j,k): $\quad (j,k) \textit{ is down} \quad \longrightarrow \quad (j,k) \textit{ is up}$, if $R.(j,k)$ then $col.(j,k) := red$

Program Invariant

$$
\begin{array}{lcl}
S & = & T\ \wedge\ (\forall j :: S1.j\ \wedge\ S2.j\ \wedge\ S3.j), \quad \text{where} \\
S1.j & = & (col.j = red \quad \Leftarrow \quad (p.j \notin Adj.j \cup \{j\}\ \vee\ col.(p.j) = red\ \vee\ col.(j,(p.j)) = red)) \\
S2.j & = & col.j = green \\
S3.j & = & (\forall k : k \in Adj.j\ \Rightarrow\ root.j = root.k)
\end{array}
$$

Program Actions

Action 1 of j :

$col.j = green\ \wedge$

$(p.j \notin Adj.j \cup \{j\}\ \vee\ col.(p.j) = red$

$\vee\ col.(j,(p.j)) = red) \quad \longrightarrow \quad col.j := red$

Action 2 of j :

$col.j = red\ \wedge$

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

Action 3 of j for k :

$col.j = green\ \wedge$

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

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

5 Related Work and Concluding Remarks

Many distributed programs for spanning tree construction have been appeared in the literature. Of these, a program by Gallagher et al [5] is especially notable, but it is fault-intolerant; i.e., it does not solve the spanning tree reconfiguration problem. More recently, stabilizing programs (i.e. nonmasking fault-tolerant programs whose T is the predicate *true*) for spanning tree reconfiguration have been presented [6, 7] that tolerate failures as well as repairs of both nodes and edges, but these programs are significantly more complex than RST, since they allow for the formation of transient cycles in the graph of the parent variables. Moreover, in the average case, these programs converge much slower than RST. Other programs we are aware of are at best tolerant only to the failures of edges or only the failures and repairs of nodes.

We measure the time complexity of convergence of RST from T to S in terms of rounds [7]. A round is a minimal, nonempty sequence of program steps wherein for each up node there exists a step where the node either executes an action or has no actions whose guards are true before or after the step. We show, in Appendix 1, that T converges to $T \wedge (\forall j :: S1.j)$ in RST within N rounds; $T \wedge (\forall j :: S1.j)$ converges to $T \wedge (\forall j :: S1.j \wedge S2.j)$ in RST within N rounds; and $T \wedge (\forall j :: S1.j \wedge S2.j)$ converges to S in RST within $2N - 3$ rounds, where N is the number of up nodes. It follows that the time complexity of RST is $O(N)$ rounds. (We remark that the proof of convergence that we present in Appendix 1 does not depend on the assumption of fairness for execution of actions; we have thus far assumed fairness only to simplify the exposition.)

Given that each process has to maintain the p variable, it is easy to see that the space complexity of RST, $O(N\ log(N))$, is optimal.

Observe that program RST can also be viewed as an nonmasking fault-tolerant solution to the leader election problem.

Finally, we note that there exist refinements of RST that yield read/write atomicity programs that are nonmasking fault-tolerant as well. We conjecture that the resulting read/write atomicity programs can be nicely translated into nonmasking fault-tolerant programs that communicate via message passing.

References

1. A. Arora and M. G. Gouda, "Closure and convergence: A foundation of fault-tolerant computing". *IEEE Trans. on Soft. Engg.* 19(11) (1993) 1015–1027
2. A. Arora, M. G. Gouda, and G. Varghese, "Constraint satisfaction as a basis for designing nonmasking fault-tolerance". *J. High Speed Networks* (1994 to appear); *Proc. 14th Intl. Conf. on Distributed Computer Systems* (1994) 424-431
3. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall (1976)
4. D. Gries, *The Science of Programming*, Springer-Verlag (1981)
5. R. G. Gallagher, P. A. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees". *ACM Trans. on Prog. Lang. and Sys.* 5(1) (1983) 66–77
6. G. Varghese, "Self-stablization by local checking and correction". *Ph.D. Dissertation*, Massachusetts Institute of Technology (1992)
7. A. Arora and M. G. Gouda, "Distributed reset". *IEEE Trans. on Computers* 43(9) (1994)

Appendix 1 — Convergence Span

In this appendix, we prove in three stages that T converges to S in RST within $O(N)$ rounds.

1. T converges to $T \wedge (\forall j :: S1.j)$ in RST within N rounds:

We show by induction on $d, 0 \leq d < N$, that each up node j whose depth in its tree is at most d satisfies $S1.j$ within d rounds. It follows that a state where $T \wedge (\forall j :: S1.j)$ holds is reached within N rounds.

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

Induction case ($d \geq 0$): By the induction hypothesis, each up node j whose depth in its tree is at most $d-1$ satisfies $S1.j$ within $d-1$ rounds. If j executes no action during round d then the negation of the first guard, i.e. $S1.j$, 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 $p.j \in Adj.j \cup \{j\} \ \wedge \ col.(p.j) \neq red$ is established. It follows that each up node j whose depth in its tree is at most d satisfies $S1.j$ within d rounds.

2. $T \wedge (\forall j :: S1.j)$ converges to $T \wedge (\forall j :: S1.j \ \wedge \ S2.j)$ in RST within N rounds:

Observe that at each state where $T \wedge (\forall j :: \ S1.j)$ holds, each up node can only execute its Action 2 or 3 and that both of these actions preserve this predicate. We show by induction on $d, 0 \leq d < N$, that each up node j whose depth in its tree is more than $N-d$ satisfies $S2.j$ within d rounds. It follows that a state where $T \wedge (\forall j :: S1.j \ \wedge \ S2.j)$ holds is reached within N rounds.

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

Induction case ($d > 0$): By the induction hypothesis, each up node j whose depth in its tree is more than $N-d+1$ is colored green within $d-1$ rounds. If j executes no action during round d then, since $T \wedge (\forall j :: S1.j)$ holds, $S2.j$ 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 $S2.j$ is preserved. It follows that each up node j whose depth in its tree is at most d satisfies $S2.j$ within d rounds.

3. $T \wedge (\forall j :: S1.j \ \wedge \ S2.j)$ converges to S in RST within $2N-3$ rounds:

Observe that at each state where $T \wedge (\forall j :: \ S1.j \ \wedge \ S2.j)$ holds, each up node can only execute its Action 3 and that Action 3 preserves this predicate. We show by induction on e, $0 \leq e < N$, that for all n, $1 \leq n < N$, each up node j that is at a distance e in the graph of the adjacency relation from some up node whose *root* value is maximum, satisfies $(Sum\, z : z > root.j \ \wedge \ (\exists k :: root.k = z) : 1) \ < \ N-n$ within $e+n-1$ rounds. It follows that all up nodes j satisfy $(Sum\, z : z > root.j \wedge (\exists k :: root.k = z) : 1) \ < \ 1$ within $2N-3$ rounds; in other words, all up nodes j have the maximum *root* value, and hence a state where S holds is reached within $2N-3$ rounds.

<u>Base case</u> $(e = 0)$: Each maximum *root* value remains the maximum in every round, so the claim is trivially true.

<u>Induction case</u> $(e > 0)$: By the induction hypothesis, for all n, each up node l at a distance of at most $e-1$ satisfies $(Sum\, z : z > root.l \ \wedge \ (\exists k :: root.k = z) : 1) \ < \ N-n$ within $(e-1)+n-1$ rounds. (*)

We proceed by induction on n.

<u>Base case</u> $(n=1)$: By (*), there exists a node l adjacent to j that satisfies $(Sum\, z : z > root.l \wedge (\exists k :: root.k = z) : 1) \ < \ N-1$ within $e-1$ rounds. It follows that j satisfies $(Sum\, z : z > root.j \ \wedge \ (\exists k :: root.k = z) : 1) \ < \ N-1$ within e rounds.

<u>Induction case</u> $(n>1)$: By (*), there exists a node l adjacent to j that satisfies $(Sum\, z : z > root.l \ \wedge \ (\exists k :: root.k = z) : 1) \ < \ N-n$ within $(e-1)+(n-1)$ rounds. By the hypothesis of the nested induction, j satisfies $(Sum\, z : z > root.j \ \wedge \ (\exists k :: root.k = z) : 1) \ < \ N-n-1$ within $e+(n-1)-1$ rounds. It follows that j satisfies $(Sum\, z : z > root.j \ \wedge \ (\exists k :: root.k = z) : 1) \ < \ N-n$ within $e+n-1$ rounds.