Distributed Reset

Anish ARORA *Mohamed GOUDA*

Department of Computer Sciences, The University of Texas at Austin[†]

Microelectronics and Computer Technology Corporation, Austin, TX, USA

Abstract

We design a reset subsystem that can be embedded in an arbitrary distributed system in order to allow the system processes to reset the system when necessary. Our design is layered, and comprises three main components: a leader election, a spanning tree construction, and a diffusing computation. Each of these components is self-stabilizing in the following sense. If the coordination between the up processes in the system is ever lost (due to failures or repairs of processes and channels) then each component eventually reaches a state where coordination is regained. This capability makes our reset subsystem very robust: it can tolerate fail-stop failures and repairs of processes and channels even when a reset is in progress.

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–*verification, fault-tolerance* ; G.2.2 [Discrete Mathematics]: Graph theory–*trees, graph algorithms.*

General Terms: Reliability, Algorithms.

Additional Key Words and Phrases: Fault-tolerance, self-stabilization, leader election, spanning tree, diffusing computation.

[†]Email Address: anish@cs.utexas.edu, gouda@cs.utexas.edu

1 Introduction

We describe in this paper how to "augment" an arbitrary distributed system so that each of its processes can reset the system to a predefined global state, when deemed necessary. The augmentation does not introduce new processes or new communication channels to the system. It merely introduces additional modules to the existing processes. The added modules, communicating with one another over existing channels, comprise what we call the reset subsystem.

Ideally, resetting a distributed system to a given global state implies resuming the execution of the system starting from the given state. With this characterization, however, each reset of a distributed system can be achieved only by a "global freeze" of the system. This seems rather limiting and, in many applications, more strict than needed. Therefore, we adopt the following, more lax, characterization: resetting a distributed system to a given global state implies resuming the execution of the system from a global state that is reachable, by some system computation, from the given global state.

There are many occasions in which it is desirable for some processes in a distributed system to initiate resets; for example,

- *Reconfiguration*: When the system is reconfigured, for instance, by adding processes or channels to it, some process in the system can be signaled to initiate a reset of the system to an appropriate "initial state".
- *Mode Change* : The system can be designed to execute in different modes or phases. If this is the case, then changing the current mode of execution can be achieved by resetting the system to an appropriate global state of the next mode.
- *Coordination Loss*: When a process observes unexpected behavior from other processes, it recognizes that the coordination between the processes in the system has been lost. In such a situation, coordination can be regained by a reset.
- *Periodic Maintenance*: The system can be designed such that a designated process periodically initiates a reset as a precaution, in case the current global state of the system has deviated from the global system invariant.

As processes and channels can fail while a reset is in progress, we are led to designing a reset subsystem that is fault-tolerant. In particular, our reset subsystem can tolerate the loss of coordination between different processes in the system (which may be caused by transient failures or memory loss) and, also, can tolerate the fail-stop failures and subsequent repairs of processes and channels.

The ability to regain coordination when lost is achieved by making the reset subsystem *self-stabilizing* in the following sense. If the reset subsystem is at a global state in which coordination between processes is lost, then the reset subsystem is guaranteed to reach, within a finite number of steps, a global state in which coordination is restored. Once coordination is restored, it is maintained unless a later failure causes it to be lost again, and the cycle repeats [6, 7]. The ability to tolerate fail-stop failures and subsequent repairs of processes and channels is achieved by allowing each process and channel in the system to be either "up" or "down" and by ensuring that the ability of the system to self-stabilize is not affected by which processes or channels are "up" or "down".

Our reset subsystem is designed in a simple, modular, and layered manner. The design consists of three major components: a leader election, a spanning tree construction, and a diffusing computation. Each of these components is self-stabilizing, can tolerate process and channel failures and repairs, and admits bounded-space implementations. These features distinguish our design of these components from earlier designs [1, 9, 10] and redress the following comment made by Lamport and Lynch [15, page 1193] : "A self-stabilizing algorithm [that translates a distributed system designed for a fixed but arbitrary network into one that works for a changing network] using a finite number of identifiers would be quite useful, but we know of no such algorithm."

The rest of the paper is organized as follows. In the next section, we describe the layered structure of our reset subsystem. This structure consists of three layers: a (spanning) tree layer, a wave layer, and an application layer. These three layers are discussed in Sections 3, 4, and 5 respectively. In Section 6, we discuss implementation issues; in particular, we exhibit bounded, low atomicity implementations of each layer. Finally, we make concluding remarks in Section 7.

2 Layers of the Reset Subsystem

We make the following assumptions concerning the distributed system to be augmented by our reset subsystem. The system consists of K processes named $P.1$, ... , $P.K$. At each instant, each process is either *up* or *down*, and there is a binary, irreflexive, and symmetric relation defined over the up processes. We call this relation the *adjacency* relation. Only adjacent processes can communicate with one another.

The set of up processes and the adjacency relation defined over them can change with time. For simplicity, however, we assume that the adjacency relation never partitions the up processes in the system. (Clearly, if partitioning does occur, then any reset request initiated in a partition will result in resetting the state of only that partition.)

Each process $P.i$ in the system consists of two modules *adj.i* and *appl.i*; see Figure 0a. The task of module *adj.i* is to maintain a set $N.i$ of the indices of all up processes adjacent to $P.i$. (Details of the implementation of *adj.i* are outside the scope of this paper. One possible implementation, however, is for each *adj.i* to communicate periodically with the *adj.j* module of every potentially adjacent process *P.j* and to employ a timeout to determine whether the index *j* of process *P.j* should be in *N.i*.) The task of the other module, *appl.i*, is application specific. To perform its task, *appl.i* can communicate with module *appl.j*, $j \neq i$, only if *j* is in *N.i*. One state of *appl.i* is distinguished. Together, the distinguished states of each *appl.i* module comprise the predefined global "reset" state of the distributed system.

Augmenting such a distributed system with a reset subsystem consists of adding two modules, *tree.i* and *wave.i*, to each process *P.i* in the system; see Figure 0b. The *tree.i* modules of adjacent processes communicate in order to maintain a rooted spanning tree that involves all the up processes in the system. (Henceforth, the two terms "process" and "up process" are used interchangeably.) The constructed tree is maintained to be consistent with the current adjacency relation of the system; thus, any changes in the adjacency relation are eventually followed by corresponding changes in the spanning tree. Each *tree.i* module keeps the index of its "father" process, *f.i*, in the maintained tree; this information is used by the local *wave.i* module in executing a distributed reset.

A distributed reset is executed by the *wave.i* modules in three phases or "waves". In the first phase, some *appl.i* requests a system reset from its local *wave.i* which forwards the request to the root of the spanning tree. If other reset requests are made at other processes, then these requests are also forwarded to the root process. It is convenient to think of all these requests as forming one "request wave". In the second phase, module *wave.i* in the root process receives the request wave, resets the state of its local *appl.i* to the state of *appl.i* in the predefined global state, and initiates a "reset wave". The reset wave travels towards the leaves of the spanning tree and causes the *wave.j* module of each encountered process to reset the state of its local *appl.j* to the state of *appl.j* in the predefined global state. When the reset wave reaches a leaf process it is reflected as a "completion wave" that travels back to the root process; this wave comprises the third phase. Finally, when the completion wave reaches the root, the reset is complete, and a new request wave can be started whenever some *appl.i* deems necessary.

From the above description, it follows that the states of different *appl.i* modules are reset at different times within the same distributed reset. This can cause a problem if some *appl.i* whose state has been reset communicates with an adjacent *appl.j* whose state has not yet been reset. To avoid this problem, we provide a session number *sn.i* in each *appl.i*. In a global state, where no distributed reset is in progress, all session numbers are equal. Each reset of the state of *appl.i* is accompanied by incrementing *sn.i*. We then require that no two adjacent *appl.i* modules communicate unless they have equal session numbers. This requirement suffices to ensure our characterization of a distributed reset; that is, a distributed reset to a given global state yields a global state that is reachable, by some system computation, from the given global state.

The *tree.i* modules in different processes constitute the tree layer discussed in Section 3. The *wave.i* modules constitute the wave layer discussed in Section 4. The *appl.i* modules constitute the application layer discussed in Section 5.

2.1 Programming Notation

The program of each process has the form

```
begin ⟨module⟩ ⫿ ... ⫿ ⟨module⟩ end
```

Each module is of the form

```
module ⟨module name⟩
    var ⟨variable declarations⟩ ;
    parameter ⟨parameter declarations⟩ ;
    begin
        ⟨action⟩ ⫿ ... ⫿ ⟨action⟩
    end
```

Thus, a module of a process is defined by a set of variables, a set of parameters, and a set of actions. Each of these is defined in some detail next.

Each variable in the variable set of a module can be updated (i.e., written) only by modules in that process; each variable can be read only by modules in that process and modules in adjacent processes.

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

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

Each action in the action set of a module has the form

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

A guard is a boolean expression over the variables and parameters in the module, and the variables of one adjacent process. An assignment statement updates one or more variables in the module.

The operational semantics for a system of such processes is as follows. A *state* of the system is defined by a value for every variable in the processes of the system. An action whose guard is true at some state of the system is said to be *enabled* at that state. A computation of the system is a maximal, fair sequence of system steps: in each step, some action that is enabled at the current state is executed, thereby yielding the next state in the computation. The maximality of a computation implies that no computation is a proper prefix of another computation. The fairness of a computation means that each continuously enabled action is eventually executed in the computation [12].

3 The Tree Layer

The task of the tree layer is to continually maintain a rooted spanning tree even when there are changes in the set of up processes or in the adjacency relation. In the solution described below, we accommodate such changes by ensuring that the tree layer performs its task irrespective of which state it starts from.

In our solution, the rooted spanning tree is represented by a "father" relation between the processes. Each *tree.i* module maintains a variable $f.i$ whose value denotes the index of the current father of process $P.i$. Since the layer can start in any state, the initial graph of the father relation (induced by the initial values of the $f.i$ variables) may be arbitrary. In particular, the initial graph may be a forest of rooted trees or it may contain cycles.

For the case where the initial graph is a forest of rooted trees, all trees are collapsed into a single tree by giving precedence to the tree whose root has the highest index. This is achieved as follows. Each *tree.i* module maintains a variable *root.i* whose value denotes the index of the current root process of $P.i$. If *root.i* is lower than *root.j* for some adjacent process $P.j$ then *tree.i* sets *root.i* to *root.j* and makes $P.j$ the father of $P.i$.

For the case where the initial graph has cycles, each cycle is detected and removed by using a bound on the length of the path from each process to its root process in the spanning tree. This is achieved as follows. Each *tree.i* module maintains a variable $d.i$ whose value denotes the length of a shortest path from $P.i$ to $P.(root.i)$. To detect a cycle, *tree.i* sets $d.i$ to be $d.(f.i){+}1$ whenever $f.i \in N.i$ and $d.i < K$. The net effect of executing this action is that if a cycle exists then the $d.i$ value of each process $P.i$ in the cycle gets "bumped up" repeatedly. Eventually, some $d.i$ exceeds $K-1$, where K is the maximum possible number of up processes. Since the length of each path in the adjacency graph is bounded by $K{-}1$, the cycle is detected. To remove a cycle that it has detected, *tree.i* makes $P.i$ its own father.

Because of our assumption that the initial state is arbitrary, we need to consider all other cases where the initial values of $f.i$, *root.i* and $d.i$ are inconsistent. One possibility is that these initial values are "locally" inconsistent, that is, one or more of the following hold: $root.i < i$, $f.i = i$ but $root.i \neq i$ or $d.i \neq 0$, or $f.i$ is not i nor in $N.i$. In this case, *tree.i* makes itself locally consistent by setting *root.i* to i, $f.i$ to i and $d.i$ to 0.

Another possibility is that *root.i* may be inconsistent with respect to the state of the father process of $P.i$, that is, $root.i \neq root.(f.i)$ may hold. In this last case, *tree.i* corrects the value of *root.i* to that of *root.j*.

Module *tree.i* is given in Figure 1.

```
module      $tree.i\ (i : 1\ ..\ K)$
var         $root.i, f.i : 1\ ..\ K;$
            $d.i$ : integer;
parameter   $j : 1\ ..\ K;$

begin

        $(root.i < i)\ \vee$
        $(f.i = i\ \wedge\ (root.i \neq i\ \vee\ d.i \neq 0))\ \vee$
        $(f.i \notin (N.i \cup \{i\})\ \vee\ d.i \geq K)$        $\longrightarrow$  $root.i, f.i, d.i := i, i, 0$
    ⫿
        $f.i = j\ \wedge\ j \in N.i\ \wedge\ d.i < K\ \wedge$
        $(root.i \neq root.j\ \vee\ d.i \neq d.j+1)$        $\longrightarrow$  $root.i, d.i := root.j, d.j+1$
    ⫿
        $(root.i < root.j\ \wedge\ j \in N.i\ \wedge\ d.j < K)\ \vee$
        $(root.i = root.j\ \wedge\ j \in N.i\ \wedge\ d.j+1 < d.i)$        $\longrightarrow$  $root.i, f.i, d.i := root.j, j, d.j+1$

end
```

Figure 1: Module *tree.i*

We show in Appendix A that starting at any state (i.e., one that could have been reached by any number of changes in the set of up processes and the adjacency relation over them), the tree layer is guaranteed to eventually reach a state satisfying the state predicate G, where

$$\begin{aligned} G \ \equiv\ & (k = \underline{max}\,\{i \mid P.i \text{ is up}\}) \ \ \wedge \\ & (\forall i : P.i \text{ is up} : \\ & \quad (i = k \ \Rightarrow\ (root.i = i \ \wedge\ f.i = i \ \wedge\ d.i = 0)) \ \ \wedge \\ & \quad (i \neq k \ \Rightarrow\ (root.i = k \wedge (\exists j : j \in N.i : f.i = j \wedge d.i = d.j+1 \wedge d.j = \underline{min}\{d.j' | j' \in N.i\})))) \end{aligned}$$

At each state in G, for each process $P.i$, $root.i$ equals the highest index among all up processes, $f.i$ is such that some shortest path between process $P.i$ and the root process $P.(root.i)$ passes through the father process $P.(f.i)$, and $d.i$ equals the length of this path. Therefore, a rooted spanning tree exists. Also, note that each state in G is a fixed-point; i.e., once the *tree.i* modules reach a state in G, no action in any of the *tree.i* modules is enabled.

Our proof employs the "convergence stair" method [13]: we exhibit a finite sequence of state predicates $H.0, H.1, ..., H.K$ such that

(i) $H.0 \equiv true$

(ii) $H.K \equiv G$

(iii) For each l such that $0 \leq l \leq K$:
$H.l$ is closed under system execution; that is, once $H.l$ holds in an arbitrary system computation, it continues to hold subsequently.

(iv) For each l such that $0 \leq l < K$:
Upon starting at an arbitrary state in $H.l$ the system is guaranteed to reach a state in $H.(l+1)$.

We also show that convergence to a state in G occurs within $O(K + (deg \times dia))$ rounds, where deg is the maximum degree of nodes in the adjacency graph, dia is the diameter of the adjacency graph and, informally speaking, a round is a minimal sequence of system steps wherein each process attempts to execute at least one action.

We conclude this section with the remark that the problems of leader election and spanning tree construction have received considerable attention in the literature (see, for example, [15, 16, 17]). Most of these algorithms are based on the assumption that all processes start execution in some designated initial state. This restriction is too severe for our purposes, and we have lifted it by designing the tree layer to be self-stabilizing; i.e., insensitive to the initial state. We note that a self-stabilizing spanning tree algorithm has been recently described in [9]. However, the algorithm in [9] is based on the simplifying assumption that, at all times, there exists a special process which knows that it is the root. We have not made this assumption: if a root process fails, then the remaining up processes elect a new root.

4 The Wave Layer

As outlined in Section 2, the task of the wave layer is to perform a diffusing computation [10] in which each *appl.i* module resets its state. The diffusing computation uses the spanning tree maintained by the tree layer, and consists of three phases. In the first phase, some *appl.i* module requests its local *wave.i* to initiate a global reset; the request is propagated by the wave modules along the spanning tree path from process *P.i* to the tree root *P.j*. In the second phase, module *wave.j* in the tree root resets the state of its local *appl.j* and initiates a reset wave that propagates along the tree towards the leaves; whenever the reset wave reaches a process *P.k* the local *wave.k* module resets the state of its local *appl.k* . In the third phase, after the reset wave reaches the tree leaves it is reflected as a completion wave that is propagated along the tree to the root; the diffusing computation is complete when the completion wave reaches the root.

To record its current phase, each *wave.i* module maintains a variable *st.i* that has three possible values: *normal*, *initiate*, and *reset*. When *st.i* = *normal*, module *wave.i* has propagated the completion wave of the last diffusing computation and is waiting for the request wave of the next diffusing computation. When *st.i* = *initiate*, module *wave.i* has propagated the request wave of the ongoing diffusing computation and is waiting for its reset wave. When *st.i* = *reset*, module *wave.i* has propagated the reset wave of the ongoing diffusing computation and is waiting for its completion wave.

Variable *st.i* is updated as follows. To initiate a new diffusing computation, the local *appl.i* module updates *st.i* from *normal* to *initiate*. To propagate a request wave, *wave.i* likewise updates *st.i* from *normal* to *initiate*. To propagate a reset wave, *wave.i* updates *st.i* from a value other than *reset* to *reset*. Lastly, to propagate a completion wave, *wave.i* updates *st.i* from *reset* to *normal*.

It is possible for some *appl.i* to update *st.i* from *normal* to *initiate* before the completion wave of the last diffusing computation reaches the root process; thus, multiple diffusing computations can be in progress simultaneously. To distinguish between successive diffusing computations, each *wave.i* module maintains an integer variable *sn.i* denoting the current session number of *wave.i*.

Recall that the operation of the wave layer is subject to changes in the set of up processes and in the adjacency relation. As before, we accommodate such changes by ensuring that the layer performs its task irrespective of which state it starts from. In our solution, starting from an arbitrary state, the wave layer is guaranteed to reach a steady state where all the *sn.i* values are equal and each *st.i* has a value other than *reset*. In particular, if no diffusing computation is in progress in a steady state, then all the *sn.i* values are equal and each *st.i* has the value *normal*. Furthermore, if a diffusing computation is initiated in a steady state where all $sn.i$ have the value m then it is guaranteed to terminate in a steady state where all $sn.i = m+1$. This is achieved by requiring that, during the reset wave, each *wave.i* module increments $sn.i$ when it resets the state of the local *appl.i* module.

Module *wave.i* is given in Figure 2. The module has five actions. Action (1) propagates the request wave from a process to its father in the spanning tree. When the request wave reaches the root process, action (2) starts a reset wave at the root process. Action (3) propagates the reset wave from the father of a process to the process. Action (4) propagates the completion wave from the children of a process to the process.

The above four actions of all *wave.i* modules collectively perform a correct diffusing computation provided that the wave layer is in a steady state. The steady states of the wave layer are those where each *wave.i* satisfies $Gd.i$,

$$\begin{aligned} Gd.i \equiv\ & ((f.i=j\ \wedge\ st.j\neq reset)\ \Rightarrow\ (st.i\neq reset\ \wedge sn.j=sn.i))\ \wedge \\ & ((f.i=j\ \wedge\ st.j=reset)\ \Rightarrow\ ((st.i\neq reset\ \wedge sn.j=sn.i+1)\ \vee\ sn.j=sn.i))\ . \end{aligned}$$

Action (5) ensures the self-stabilization of the wave layer to steady states.

```
module      $wave.i\ (i : 1 .. K)$
var         $sn.i$ : integer;
            $st.i : \{normal\ ,\ initiate\ ,\ reset\}$;
parameter   $j : 1 .. K$;

begin

        $st.i = normal \wedge f.j = i \wedge j \in N.i \wedge st.j = initiate$   $\longrightarrow$   $st.i := initiate$   (1)
    []
        $st.i = initiate \wedge f.i = i$   $\longrightarrow$   $st.i, sn.i := reset, sn.i + 1$   (2)
                                              ; {reset $appl.i$ state}
    []
        $st.i \neq reset \wedge f.i = j \wedge st.j = reset \wedge sn.i + 1 = sn.j$   $\longrightarrow$   $st.i, sn.i := reset, sn.j$   (3)
                                              ; {reset $appl.i$ state}
    []
        $st.i = reset \ \wedge$
        $(\forall j \in N.i :\ (f.j = i) \Rightarrow (st.j \neq reset\ \wedge sn.i = sn.j))$   $\longrightarrow$   $st.i := normal$   (4)
    []
        $\neg Gd.i$   $\longrightarrow$   $st.i, sn.i := st.j, sn.j$   (5)

end
```

Figure 2: Module *wave.i*

We show in Appendix B that starting at any state, the wave layer is guaranteed to eventually reach a steady state satisfying $(\forall i : sn.i = n \ \wedge \ st.i \neq reset)$ for some integer n. Our proof of this consists of showing that

(i) Starting at an arbitrary state, the system is guaranteed to reach a state in GD, where $GD \equiv (\forall i : (p.i \text{ is up}) \Rightarrow \ Gd.i)$.

(ii) The state predicate GD is closed under system execution.

(iii) Starting at an arbitrary state in GD where the root process $P.k$ has $sn.k = n$, the system is guaranteed to reach a state in $(\forall i : sn.i = n \ \wedge \ st.i \neq reset)$.

We also show that each diffusing computation that is initiated at a state in GD will terminate; i.e., starting from a state satisfying $(GD \ \wedge \ (\exists i : sn.i = n \ \wedge \ st.i = initiate))$, for some integer n, the system is guaranteed to reach a state in $(GD \ \wedge \ (\forall i : sn.i = n+1 \ \wedge \ st.i \neq reset))$.

Lastly, we show that convergence to a GD state occurs within $O(ht)$ rounds and that diffusing computations terminate within $O(\,min\,(ht{\times}dg, n)\,)$ rounds, where ht is the height of the spanning tree constructed by the tree layer, dg is the maximum degree of nodes in the spanning tree, and n is the number of up processes in the system.

5 The Application Layer

The application layer in a given distributed system is composed of the *appl.i* modules as shown in Figure 0. In this section, we discuss two modifications to the application layer by which our reset subsystem can be correctly added to the given distributed system.

The first modification is to augment each *appl.i* module with actions that allow it to request a distributed reset; as discussed in Section 4, these actions set the variable *st.i* to *initiate* and are enabled when $st.i = normal$ holds and a distributed reset is necessary. The situations in which distributed resets are necessary are application specific. One such situation, however, is when the global state of the application layer is erroneous. Erroneous states may be detected by periodically executing a self-stabilizing global state detection algorithm [8, 14]. Towards this end, we note that it is possible to implement a self-stabilizing global state detection with minor modifications to our reset subsystem.

The second modification is to restrict the actions of each *appl.i* module so that the application layer can continue its execution while a distributed reset is in progress. (Recall that one objective of our design is to avoid freezing the execution of the given distributed system while performing resets.) This modification is based on the observation that, during a distributed reset, *appl.i* modules can continue executing their actions as long as there is no communication between modules one of which has been reset and another which has not been reset. Equivalently, if *appl.i* modules communicate they should have the same session number (*sn*) values. Therefore, we require that the expression "$sn.i = sn.j$" be conjoined to the guard of each *appl.i* action that accesses a variable updated by *appl.j*, $i \neq j$. The net effect of this modification is that upon completion of a distributed reset the collective state of all *appl.i* modules is reachable by some application layer execution from the given collective state that the *appl.i* modules are reset to.

6 Implementation Issues

In this section, we discuss two issues related to implementations of modules *tree.i* and *wave.i* . First, we show that the state-space of each process can be bounded and, second, we show how to refine the "high" atomicity actions employed thus far into "low" atomicity ones.

6.1 Bounded-Space Construction

Each *tree.i* module, $i \in \{1 \ldots K\}$, updates three variables each requiring $log\, K$ bits. In contrast, module *wave.i* uses an unbounded session number variable. A bounded construction is also possible: *wave.i* can be transformed by making *sn.i* of type $\{0..N-1\}$, where N is an arbitrary natural constant greater than 1, and replacing the increment operation in the first action with an increment operation in *modulo* N arithmetic. Thus, each *wave.i* module can be implemented using a constant number of bits. The proof of correctness of the transformed module is similar to the proof presented in Appendix B, and is left to the reader.

6.2 Transformation to Read/Write Atomicity

Thus far, our design of the *tree.i* and *wave.i* modules has not taken into account any atomicity constraints. Some actions in these modules are of high atomicity; these actions read variables updated by other processes and instantaneously write other variables. We now refine our design so as to implement these modules using low atomicity actions only.

Consider the following transformation. For each variable $x.i$ updated by process $P.i$, introduce a local variable $\tilde{x}.j.i$ in each process $P.j, j \neq i$, that reads $x.i$. Replace every occurrence of $x.i$ in the actions of $P.j$ with $\tilde{x}.j.i$, and add the read action $\tilde{x}.j.i := x.i$ to the actions of $P.j$. Based on this transformation, read/write atomicity modules for *tree.i* and *wave.i* are presented next, along with proofs of correctness.

The code for read/write atomicity implementation of module *tree.i* is shown in Figure 3.

We show in Appendix C that starting at any state, the tree layer is guaranteed to eventually reach a state satisfying the state predicate $\mathcal{G}$, where

$$\begin{aligned}\mathcal{G} \equiv\ & (root.k = k \ \wedge\ f.k = k \ \wedge\ d.k = 0) \ \wedge \\ & (\forall i : i \neq k \ \Rightarrow\ (root.i = k \wedge (\exists j : j \in N.i \wedge f.i = j \wedge d.i = d.j{+}1 \wedge d.j = \underline{min}\{d.j' | j' \in N.i\}))) \ \wedge \\ & (\forall i : j \in N.i \ \Rightarrow\ (\tilde{root}.i.j = k \ \wedge\ \tilde{f}.i.j = f.j \ \wedge\ \tilde{d}.i.j = d.j))\end{aligned}$$

The structure of our proof is identical to the proof presented in Appendix A; we exhibit a finite sequence of state predicates $\mathcal{H}.0, \mathcal{H}.1, ..., \mathcal{H}.K$ such that

(i) $\mathcal{H}.0 \equiv true$

(ii) $\mathcal{H}.K \equiv \mathcal{G}$

(iii) For each l such that $0 \leq l \leq K$:
$\mathcal{H}.l$ is closed under system execution; that is, once $\mathcal{H}.l$ holds in an arbitrary system computation, it continues to hold subsequently.

(iv) For each l such that $0 \le l < K$:
Upon starting at an arbitrary state in $\mathcal{H}.l$ the system is guaranteed to reach a state in $\mathcal{H}.(l+1)$.

```
module      $tree.i\ (i : 1 \,..\, K)$
var         $root.i, f.i : 1 \,..\, K;$
            $d.i :$ integer;
            $\tilde{root}.i.j, \tilde{f}.i.j : 1 \,..\, K;$
            $\tilde{d}.i.j :$ integer;
parameter   $j : 1 \,..\, K\ ,\ j \neq i;$

begin

        $(root.i < i)\ \vee$
        $(f.i = i\ \wedge\ (root.i \neq i \vee d.i \neq 0))\ \vee$
        $(f.i \notin (N.i \cup \{i\})\ \vee\ d.i \geq K)$          $\longrightarrow$  $root.i, f.i, d.i := i, i, 0$
   []
        $f.i = j\ \wedge\ j \in N.i\ \wedge\ \tilde{d}.i < K\ \wedge$
        $(root.i \neq \tilde{root}.i.j\ \vee\ d.i \neq \tilde{d}.i.j+1)$          $\longrightarrow$  $root.i, d.i := \tilde{root}.i.j, \tilde{d}.i.j+1$
   []
        $(root.i < \tilde{root}.i.j\ \wedge\ j \in N.i\ \wedge\ \tilde{d}.i.j < K)\ \vee$
        $(root.i = \tilde{root}.i.j\ \wedge\ j \in N.i\ \wedge\ \tilde{d}.i.j+1 < d.i)$          $\longrightarrow$  $root.i, f.i, d.i := \tilde{root}.i.j, j, \tilde{d}.i.j+1$
   []
        $j \in N.i \wedge (root.j \neq \tilde{root}.i.j \vee f.j \neq \tilde{f}.i.j \vee d.j \neq \tilde{d}.i.j)$ $\longrightarrow$  $\tilde{root}.i.j, \tilde{f}.i.j, \tilde{d}.i.j := root.j, f.j, d.j$

end
```

Figure 3: Implementation of *tree.i* using Read/Write Atomicity

---

The code for read/write atomicity implementation of module *wave.i* is shown in Figure 4.

We show in Appendix D that starting at any state, the wave layer is guaranteed to eventually reach a state satisfying $(\forall i : sn.i = n\ \wedge\ st.i \neq reset)$ for some integer n. The structure of our proof is identical to the proof presented in Appendix B; we exhibit a state predicate $\mathcal{GD}$ such that

(i) Starting at an arbitrary state, the system is guaranteed to reach a state in $\mathcal{GD}$.

(ii) $\mathcal{GD}$ is closed under system execution.

```
module      $wave.i\ (i : 1 .. K)$
var         $st.i : \{normal , initiate , reset\}$;
            $sn.i$ : integer;
            $\tilde{sn}.i.j$ : integer;
            $\tilde{st}.i.j : \{normal , initiate , reset\}$;
parameter   $j : (1 .. K) , j \neq i$;

begin

        $st.i = normal \wedge \tilde{f}.i.j = i \wedge \tilde{st}.i.j = initiate$   $\longrightarrow$  $st.i := initiate$
   []
        $st.i = initiate \wedge f.i = i$   $\longrightarrow$  $st.i, sn.i := reset, sn.i + 1$ ; {reset $appl.i$ state}
   []
        $st.i \neq reset \wedge f.i = j \wedge \tilde{st}.i.j = reset \wedge sn.i \neq \tilde{sn}.i.j$   $\longrightarrow$  $st.i, sn.i := reset, \tilde{sn}.i.j$ ; {reset $appl.i$ state}
   []
        $st.i = reset \wedge$
        $(\forall j \in N.i , (\tilde{f}.i.j = i) \Rightarrow (\tilde{st}.i.j \neq reset \wedge sn.i = \tilde{sn}.i.j))$   $\longrightarrow$  $st.i := normal$
   []
        $st.i = \tilde{st}.i.j \wedge f.i = j \wedge sn.i \neq \tilde{sn}.i.j$   $\longrightarrow$  $sn.i := \tilde{sn}.i.j$
   []
        $(\tilde{f}.i.j = i \vee f.i = j) \wedge (st.j \neq \tilde{st}.i.j \vee sn.j \neq \tilde{sn}.i.j)$   $\longrightarrow$  $\tilde{st}.i.j, \tilde{sn}.i.j := st.j, sn.j$
end
```

Figure 4: Implementation of *wave.i* using Read/Write Atomicity

(iii) Starting at an arbitrary state in $\mathcal{GD}$ where the root process $P.k$ has $sn.k = n$, the system is guaranteed to reach a state in $(\forall i : sn.i = n \wedge st.i \neq reset)$.

We also show that each diffusing computation that is initiated at a state in $\mathcal{GD}$ will terminate; i.e., upon starting from a state satisfying $(\mathcal{GD} \wedge (\exists i : sn.i = n \wedge st.i = initiate))$ for some integer n the system is guaranteed to reach a state in $(\mathcal{GD} \wedge (\forall i : sn.i = n+1 \wedge st.i \neq reset))$.

We note that a similar proof exists for a bounded construction of the low atomicity *wave.i* module in which $sn.i$ is replaced with a variable of type $\{0..N-1\}$, where N is an arbitrary natural constant greater than 3, and the increment operation in the first action is replacing with an increment operation in *modulo* N arithmetic.

7 Conclusions

We have presented algorithms that enable processes in arbitrary distributed systems to perform distributed resets. These algorithms are novel in that they are self-stabilizing and can tolerate the fail-stop failures and repairs of arbitrary processes and channels even when a distributed reset is in progress.

Two comments are in order regarding our choice of fair, nondeterministic interleaving semantics. First, the requirement of fairness with respect to continuously enabled actions is not necessary, but is used only in simplifying the proofs of correctness. Second, our design remains correct even if we weaken the interleaving requirement as follows: in each step, an arbitrary subset of the processes each execute some enabled action, as long as no two executed actions access the same shared variable [2, 3, 5].

A comment is also in order regarding our methodology for achieving fault-tolerance in distributed systems. One way to achieve system fault-tolerance is to ensure that when faults occur the system continues to satisfy its input-output relation. Systems designed thus "mask" the effects of faults, and are hence said to be masking fault-tolerant. An alternative way to achieve system fault-tolerance is to ensure that when faults occur the input-output relation of the system is violated only temporarily. In other words, the system is guaranteed to eventually resume satisfying its input-output relation. In this paper, it is the latter "nonmasking" approach to fault-tolerance that we have adopted.

We give three reasons for sometimes preferring nonmasking fault-tolerance to masking fault-tolerance when designing distributed systems. First, in some distributed systems, masking fault-tolerance may be impossible to achieve. For example, there is no masking fault-tolerant distributed system whose up processes communicate asynchronously and reach consensus on a binary value even when one or more of the processes fail [11]. Second, even if it is possible to implement masking fault-tolerance, the cost of doing so may be prohibitive. For example, the amount of redundancy or synchronization required may be infeasible to implement. And third, requiring masking fault-tolerance may be more strict than is desirable. For example, a call-back telephone service that eventually establishes a connection may be quite useful even if it does not mask its initial failure to establish a connection.

Of course, to be of practical use, nonmasking fault-tolerant distributed systems should be designed so that the time taken to resume satisfying the desired input-output relation, when faults occur, is within acceptable bounds.

We envisage several applications of distributed resets where their nonmasking fault-tolerance is useful. We are currently implementing distributed operating system programs based on distributed resets including, for example, system programs for multiprocess resynchronization. We are also currently studying reconfiguration protocols for high speed networks.

We note that distributed resets provide a systematic method for making arbitrary distributed systems self-stabilizing (cf. [14]): application layer modules can be augmented to perform a self-stabilizing global state detection periodically, and to request a distributed reset upon detecting erroneous global states thereby making the distributed system self-stabilizing. Distributed resets can also be used to transform an arbitrary self-stabilizing program into an equivalent self-stabilizing program implemented in read/write atomicity.

There are several issues that need to be further investigated. One such issue is the transformation of our read/write atomicity programs (cf. Figures 3 and 4) into message passing programs, and the analysis of the resulting programs. Note that for message passing programs the predefined global reset state includes, in addition to the states of each *appl.i* module, the state of each channel in the system. Therefore, in addition to resetting the local state of the module *appl.i*, each *wave.i* module has to send some — possibly empty — sequence of application messages, each tagged with the new session number, on every outgoing channel of *P.i*.

Another issue for further study is the design of an efficient mechanism for maintaining a timely and consistent state of neighboring process indices. A third issue is the security problems involved in allowing any application process to reset the distributed system, and the protection mechanism necessary to enforce that application processes interact with the reset subsystem in the desired manner. Finally, observing that self-stabilizing systems are only one type of nonmasking fault-tolerant systems, it is desirable to investigate alternative nonmasking fault-tolerant solutions to the distributed reset problem that are less robust than our self-stabilizing solutions but are even more efficient.

Acknowledgements
We thank George Varghese for helpful discussions on this paper and the anonymous referees for their suggestions.

References

[1] Y. Afek, B. Awerbuch, and E. Gafni, "Applying static network protocols to dynamic networks", *Proceedings of 28th IEEE Symposium on Foundations of Computer Science* (1987).

[2] A. Arora, "A foundation of fault-tolerant computing," Ph.D. Dissertation, The University of Texas at Austin, December 1992.

[3] A. Arora, P. Attie, M. Evangelist, and M.G. Gouda, "Convergence of iteration systems", *Distributed Computing*, to appear.

[4] A. Arora and M.G. Gouda, "Distributed reset (extended abstract)", *Proceedings of 10th Conference on Foundations of Software Technology and Theoretical Computer Science*, *LNCS 472* (1990), pp. 316-331, Springer-Verlag.

[5] J.E. Burns, M.G. Gouda, and R.E. Miller, "On relaxing interleaving assumptions", Technical Report GIT-ICS-88/29, School of ICS, Georgia Institute of Technology (1988).

[6] G.M. Brown, M.G. Gouda, and C.-L. Wu, "Token systems that self-stabilize", *IEEE Transactions on Computers*, Vol. 38, No. 6 (1989), pp. 845-852.

[7] J.E. Burns and J. Pachl, "Uniform self-stabilizing rings", *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 2 (1989), pp. 330-344.

[8] K.M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems", *ACM Transactions on Computer Systems*, Vol. 3, No. 1 (1985), pp. 63-75.

[9] S. Dolev, A. Israeli, and S. Moran, "Self-stabilization of dynamic systems assuming only read/write atomicity", *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing* (1990), pp. 103-117.

[10] E.W. Dijkstra, and C.S. Scholten, "Termination detection for diffusing computations", *Information Processing Letters*, Vol. 11, No. 1 (1980), pp. 1-4.

[11] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process", *Journal of the ACM*, Vol. 32, No. 2 (1985), pp. 374-382.

[12] N. Francez, *Fairness*, Springer-Verlag, 1986.

[13] M.G. Gouda, and N. Multari, "Stabilizing communication protocols", *IEEE Transactions on Computers*, Vol. 40, No. 4 (1991), pp. 448-458.

[14] S. Katz and K. Perry, "Self-stabilizing extensions for message-passing systems", *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing* (1990), pp. 91-101.

[15] L. Lamport, and L. Lynch, “Distributed computing: models and methods”, *Handbook of Theoretical Computer Science*, Chapter 18, Vol. 2 (1990), pp. 1158-1199, Elsevier Science Publishers.

[16] 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.

[17] W. Tajibnapis, “A correctness proof of a topology information maintenance protocol for a distributed computer network”, *Communications of the ACM*, Vol. 20, No. 7 (1977), pp. 477-485.

8 Appendix A

Proof of Correctness for the Tree Layer.

Let dummy variables i , j , and j' range over the indices of up processes. Let $P.k$ denote the up process with highest index amongst all up processes; i.e., $k = max\{i \mid P.i \text{ is up}\}$. Let $dist.i.j$ be the length of the minimal length path from $P.i$ to $P.j$ in the adjacency graph. We define
$H.1 \equiv (root.k = k \wedge f.k = k \wedge d.k = 0) \wedge (\forall i : \ root.i \leq k \wedge (dist.i.k > 0 \Rightarrow (root.i = k \Rightarrow d.i > 0)))$

Lemma 1: The set of states $H.1$ is closed under system execution.

Proof: Our obligation is to show for each state s in $H.1$ and for each action enabled at s that executing the assignment statement of the action in s yields a state in $H.1$. We meet this obligation by first noting that the variables $root.i$, $f.i$ and $d.i$ are modified only by the actions of module $tree.i$. Second, for $i = k$, no action of $tree.i$ is enabled at s since $(root.k = k \wedge f.k = k \wedge d.k = 0)$ holds in s. Finally, for $i \neq k$, executing the assignment statement of any action of $tree.i$ that is enabled at s preserves $(root.i \leq k \ \wedge \ (dist.i.k > 0 \Rightarrow (root.i = k \ \Rightarrow \ d.i > 0)))$ since the value assigned to $root.i$ is at most k, and if that value is identically k then the value assigned to $d.i$ exceeds 0. $\Box$

Lemma 2: Upon starting at an arbitrary state, i.e., a state in $H.0$, the system is guaranteed to reach a state in $H.1$.

Proof: We first show that starting at an arbitrary state s, the system is guaranteed to reach a state in $(\forall i : \ \ root.i \leq k)$. To see this, consider the "variant" function $\#(s) = \langle m(s), md(s), num(s) \rangle$, where

$\langle m(s), md(s), num(s) \rangle$ is a sequence of three natural numbers,
$m(s) = \underline{max} \ \{ \ root.i \ \}$,
$md(s) = K - \underline{min} \ \{ \ d.i \mid root.i = m(s) \ \}$, and [‡]
$num(s) = \| \ \{ \ i \mid root.i = m(s) \ \wedge \ d.i = K - md(s) \ \} \ \|$.

<u>Remark:</u> Our proofs of progress properties will typically involve exhibiting a variant function whose range is a set of fixed length sequences of natural numbers. We define a lexical ordering $\prec$ between such sequences (of length, say, N):
$\langle x.1, x.2, ..., x.N \rangle \ \prec \ \langle y.1, y.2, ..., y.N \rangle \ \equiv$
$$(\exists n : 1 \leq n \ \wedge \ n \leq N \ \wedge \ x.n < y.n \ \wedge \ (\forall m : (1 \leq m \wedge m < n) \Rightarrow x.m = y.m))$$
Note that $\prec$ is a well-founded relation; thus, there is no infinitely descending chain of elements in the range of the variant function. (End of remark.)

Provided $k < m(s)$, the value assigned by $\#$ to s is, under system execution, *nonincreasing* with

[‡]We adopt the convention that upon application to the empty set $\underline{max}$ yields 0 and $\underline{min}$ yields ∞.

respect to $\prec$. That is, for arbitrary natural number constants M, MD and NUM the set of states s' in $(k < M \ \wedge\ \#(s') \prec \langle M, MD, NUM\rangle)$ is closed under system execution. The last claim follows from the observation that each action of *tree.i* assigns to *root.i* a value that is at most M, and if that value is identically M then the value assigned to *d.i* is strictly greater than $K{-}MD$. Moreover, provided $k < m(s)$, the value assigned by $\#$ to s is, under system execution, guaranteed to eventually *decrease* with respect to $\prec$. To see this, consider any process *P.i* such that $(\mathit{root.i} = m(s) \ \wedge\ d.i = K - md(s))$. As long as the variant function value does not change, either the first or the second action of *tree.i* is enabled. By fairness, we have that continuously enabled actions are eventually executed; thus, the variant function value eventually decreases with respect to $\prec$. As $\prec$ is a well-founded relation, the system is guaranteed to eventually reach a state s in which $k \geq m(s)$ and, therefore, $(\forall i : \mathit{root.i} \leq k)$ is true.

Next, from module code, we see that the set of states satisfying $(\forall i\ :\ \mathit{root.i} \leq k)$ is closed under system execution. Now, in an arbitrary state satisfying $(\forall i : \mathit{root.i} \leq k)$ either $(\mathit{root.k} = k \ \wedge\ f.k = k \ \wedge\ d.k = 0)$ holds or the first action of *tree.k* is enabled. By fairness, we can conclude that the first action of *tree.k* will eventually be executed yielding a state in the set $((\forall i : \mathit{root.i} \leq k) \ \wedge\ (\mathit{root.k} = k \ \wedge\ f.k = k \ \wedge\ d.k = 0))$ which, in turn, is seen to be closed under system execution.

Finally, for an arbitrary process *P.j*, $j \neq k$, some action of *tree.j* is necessarily enabled as long as the system state satisfies $((\forall i : \mathit{root.i} \leq k) \ \wedge\ (\mathit{root.k} = k \ \wedge\ f.k = k \ \wedge\ d.k = 0) \ \wedge\ (\mathit{root.j} = k \ \wedge\ d.j = 0))$. By fairness, some action of *tree.j* will eventually be executed thereby yielding a state in $((\forall i : \mathit{root.i} \leq k) \ \wedge\ (\mathit{root.k} = k \ \wedge\ f.k = k \ \wedge\ d.k = 0) \ \wedge\ (\mathit{root.j} = k \ \Rightarrow\ d.j \neq 0))$. This set of states is closed under system execution. As the argument holds for an arbitrarily chosen process *P.j*, the system is guaranteed to eventually reach a state in $H.1$. $\square$

Define by induction over l, $1 \leq l < K$,

$$\begin{aligned}
H.(l{+}1) \equiv\ & H.l \ \wedge \\
& (\forall i : \mathit{dist.i.k} = l \ \Rightarrow \\
& \quad (\mathit{root.i} = k \ \wedge \\
& \quad (\exists j : j \in N.i : f.i = j \wedge d.i = d.j{+}1 \wedge d.j = \underline{min}\{d.j' \mid \mathit{root.j'} = k \wedge j' \in N.i\}))) \ \wedge \\
& (\forall i : \mathit{dist.i.k} > l \ \Rightarrow\ (\mathit{root.i} = k \ \Rightarrow\ d.i > l))
\end{aligned}$$

Lemma 3: For each l such that $1 \leq l < K$ the following proposition holds:
$H.(l{+}1)$ is closed under system execution.

Proof: We prove by induction on l that

$$H.(l{+}1) \ \Rightarrow\ (\forall\, i\ :\ \mathit{dist.i.k} = l \ \Rightarrow\ d.i = l), \text{ and} \qquad (0)$$

$$H.(l{+}1) \text{ is closed under system execution.} \qquad (1)$$

Base Case: $l=0$.

Since $(\forall\, i \ :\ dist.i.k=0 \ \equiv\ i=k)$, and $(H.1 \ \Rightarrow\ d.k=0)$, (0) follows. Assertion (1) follows from Lemma 1.

Induction Step: $l>0$.

The induction hypothesis is

$$H.l \ \Rightarrow\ (\forall\, i \ :\ dist.i.k=(l-1) \ \Rightarrow\ d.i=(l-1)), \text{ and} \tag{2}$$

$$H.l \text{ is closed under system execution.} \tag{3}$$

A proof of (0) follows:

$H.(l+1)$

$\Rightarrow$ { from the definition of $H.(l+1)$ }

$H.(l+1) \ \wedge\ H.l$

$\Rightarrow$ { from (2) and the definition of $H.l$ }

$H.(l{+}1) \wedge (\forall i : (dist.i.k=l{-}1 \ \Rightarrow\ d.i=l{-}1) \wedge (dist.i.k>l{-}1 \ \Rightarrow\ (root.i=k \ \Rightarrow\ d.i>l{-}1)))$

$\Rightarrow$ { arithmetic }

$H.(l+1) \ \wedge\ (\forall i : dist.i.k=l \ \Rightarrow\ \underline{min}\{\, d.j' \mid root.j'=k \ \wedge\ j' \in N.i\}= l-1)$

$\Rightarrow$ { from the second conjunct of $H.(l+1)$ }

$(\forall\, i \ :\ dist.i.k=l \ \Rightarrow\ d.i=l)$

To prove (1), we note that the set of states $H.(l+1)$ is closed under system execution because

- $H.l$ is preserved under system execution according to (3),
- If $dist.i.k = l$, it follows from (0) that $d.i = l$. Also, it follows from $H.l$ that $(\forall j : (j \in N.i \ \wedge\ root.j=k) \ \Rightarrow\ (d.j \geq l-1))$. Thus, no action in module *tree.i* is enabled and the second conjunct of $H.(l+1)$ is preserved under system execution, and
- If $dist.i.k > l$, it follows from $H.l$ that $(\forall j : (j \in N.i \ \wedge\ dist.i.k > l \ \wedge\ root.j = k) \ \Rightarrow\ d.j > l-1)$. Hence, $(\forall i : dist.i.k > l \ \Rightarrow\ (root.i=k \ \Rightarrow\ d.i > l))$ is preserved under system execution. □

Lemma 4: For each l such that $1 \leq l < K$ the following proposition holds:
Upon starting at an arbitrary state in $H.l$, the system is guaranteed to reach a state in $H.(l+1)$

Proof: Consider an arbitrary process $P.i$ such that $dist.i.k = l$. At each state in $H.l$, either $(root.i=k \ \wedge\ (\exists j : j \in N.i :\ f.i=j \ \wedge\ d.i=d.j+1 \ \wedge\ d.j=\underline{min}\,\{\, d.j' \mid root.j'=k \ \wedge\ j' \in N.i\,\}))$ holds or the third action of *tree.i* is enabled for parameter j such that $dist.j.k = (l-1)$. By fairness and the fact that $H.l$ is closed under system execution, the third action will be executed eventually for such a parameter value, thereby establishing $H.l \ \wedge\ (root.i=k \ \wedge\ (\exists j : j \in N.i : f.i=j \wedge d.i=d.j{+}1 \wedge d.j=\underline{min}\{\, d.j' \mid root.j'=k \wedge j' \in N.i\,\}))$. This set of states is closed and no action of *tree.i* is enabled in it. Since $P.i$ is chosen arbitrarily, we can repeat this argument to establish that eventually the system is at some state in $H.l \ \wedge\ (\forall i : dist.i.k = l \ :\ (root.i =$

$k \wedge (\exists j : j \in N.i : f.i = j \ \wedge \ d.i = d.j + 1 \ \wedge \ d.j = \underline{min}\{ d.j' \mid root.j' = k \ \wedge \ j' \in N.i \})))$.

Next, consider an arbitrary process $P.j$ such that $dist.j.k > l$. Recall that, by definition, $H.l \Rightarrow (\forall j' : dist.j.k > l-1 : root.j' \leq k \ \wedge \ (root.j' = k \ \Rightarrow \ d.j' > l-1))$. Thus, if executing some action sets $root.j$ to k, then $d.j$ is set to a value that is greater than l. Also, if $(root.j = k \ \Rightarrow \ d.j > l)$ does not hold, then the second or third actions of *tree.j* are continuously enabled and will eventually be executed due to fairness thereby establishing $d.j > l$. Since $P.j$ is chosen arbitrarily, we can repeat this argument to establish that eventually the system is at a state where $(\forall j : dist.j.k > l : root.j = k \ \Rightarrow \ d.j > l)$ holds, and hence $H.(l+1)$ holds. $\square$

Theorem 1: {Closure of G}
The set of states G is closed under system execution.

Proof: $G \equiv H.K$. The theorem follows from Lemma 3. $\square$

Theorem 2: {Convergence to G}
Upon starting at an arbitrary state, the system is guaranteed to reach a state in G.

Proof: By transitivity, using Lemmas 2 and 4, and $G \equiv H.K$. $\square$

It now remains to analyze the rate of convergence of the system to a state in G. Recall that in any system computation, the nondeterminism in the choice of actions to be executed is constrained only by fairness. Fairness is a lax constraint in that it allows for computations wherein execution of some actions is attempted infrequently compared to other actions. Consequently, some computations may converge slowly (for example, the tree layer may converge slowly when execution of the first action of *tree.k* is attempted infrequently). To ensure quick convergence, we therefore propose to implement the following constraint on the choice of actions to be executed. For each *tree.i* module, execution of its actions involving neighboring processes is attempted in an arbitrary but fixed cyclic order; also, execution of the first action of *tree.i* is attempted once in every two consecutive attempts at executing actions of *tree.i* .

Below, we show that the system thus implemented is guaranteed to converge to a state in G within $O(K + (deg \times dia))$ rounds, where *deg* is the maximum degree of nodes in the adjacency graph, *dia* is the diameter of the adjacency graph, and a round of a computation is a minimal sequence of steps S such that each process in the system that is enabled at some state along S executes at least one action in S .

First, we show by induction that after r rounds $(0 \leq r \leq K)$ in a computation, if process $P.(root.i)$ is down then $d.i$ is at least r. The base case $(r = 0)$ is trivially true. For the induction step $(r > 0)$, we observe that $P.(root.i)$ is down iff the action that last updated the state of $P.i$ involved accessing the state of a neighbor j such that $P.(root.j)$ was down; thus $d.i$ was set to a value at least r. Hence, after K rounds, if $P.(root.i)$ is down then $d.i$ is at least K. It now follows from the actions of $tree.i$ that after $K+1$ rounds, $P.(root.i)$ is up for each process i.

Next, we show that after $deg \times dia$ more rounds, a state in G is reached. From the constraint on execution of actions, it follows that once a state is reached where for $root.i$ for each process $P.i$ is up then within the next 2 rounds a state is reached where $f.k = k \wedge d.k = 0 \wedge root.k = k$ holds. Subsequently, within the next $2 \times deg$ rounds, each neighboring process $P.i$ updates its state based on the state of $P.k$, and thus $f.i = k \wedge d.i = 1 \wedge root.i = k$ holds. Repeating this argument dia times, it follows that the system state is in G within $deg \times dia$ rounds.

Hence, the convergence rate is $O(K + (deg \times dia))$ rounds.

9 Appendix B

Proof of Correctness for the Wave Layer.

Let dummy variables i , j and j' range over the indices of up processes, and n range over the integers. Let $P.k$ denote the up process with highest index amongst all up processes; i.e., $k = max\{i \mid P.i \text{ is up}\}$.

Theorem 3: {Closure of GD}
The set of states GD is closed under the execution of the system.

Proof: The variables $st.i$ and $sn.i$ of an arbitrary process $P.i$ are modified only by
(T1) the actions of module $wave.i$, and
(T2) the action(s) of module $appl.i$ that atomically change $st.i$ from $normal$ to $initiate$, and do not change $sn.i$.

Therefore, to prove that GD is closed under system execution, it suffices to show that for each action a of type (T1) or (T2) the following Hoare triples hold:

$$\{GD \ \wedge \ \langle \text{ guard-of-a } \rangle\} \quad \langle \text{ assignment-statement-of-a } \rangle \quad \{Gd.i\} \quad , \tag{0}$$

and for all j' such that $f.j' = i$

$$\{GD \ \wedge \ \langle \text{ guard-of-a } \rangle\} \quad \langle \text{ assignment-statement-of-a } \rangle \quad \{Gd.j'\} \quad . \tag{1}$$

We meet this obligation by considering the following cases:

- Executing the first action of $wave.i$ maintains the relation $st.i \neq reset$ and does not change $sn.i$. From this (0) and (1) follow. The same argument applies to all actions of type (T2).
- The second action of $wave.i$ is enabled for $i = k$ only. $Gd.k$ is trivially true. Hence, (0) follows. The precondition of the Hoare triple in (1) implies that $(st.j' \neq reset \ \wedge \ sn.k = sn.j')$. Thus, $(st.k = reset \ \wedge \ st.j' \neq reset \ \wedge \ sn.k = sn.j' + 1)$ holds upon executing the second action, thereby establishing (1).
- Upon executing the third action, $(st.(f.i) = reset \ \wedge \ sn.(f.i) = sn.i)$ holds. Therefore, (0)

is valid. Also, the precondition of the Hoare triple in (1) implies $(st.j' \neq reset \ \wedge \ sn.i = sn.j' \ \wedge \ sn.(f.i) = sn.i+1)$ and so $(st.i = reset \ \wedge \ st.j' \neq reset \ \wedge \ sn.i = sn.j'+1)$ holds in the postcondition. This validates (1).

- When the fourth action is enabled, $(st.(f.i) = reset \ \wedge \ sn.(f.i) = sn.i)$ is necessarily true. Upon execution, this action leaves $sn.i$ unchanged. Thus, (0) is true. For (1), we note that the precondition of the Hoare triple in (1) implies $(st.j' \neq reset \ \wedge \ sn.i = sn.j')$. From this, $Gd.j'$ is seen to hold upon executing this action.
- The fifth action is not enabled at any state of GD. In this final case, (0) and (1) are trivially true. □

Theorem 4: {Convergence to GD}
Upon starting at an arbitrary state, the system is guaranteed to reach a state in GD.

Proof: Let s denote the system state. Let $anc.i.j$ denote the predicate that $P.j$ is an ancestor of $P.i$ in the spanning tree. We define a variant function $\sharp$:

$$\sharp(s) \ = \ K - \mid \{ \, i \mid Gd.i \ \wedge \ (\forall j : anc.i.j \ \Rightarrow \ Gd.j \text{ holds at } s)\} \mid$$

Elements in the range of $\sharp$ are related by the well-founded relation $\prec$ that we introduced previously.

Observe that the set of states $(Gd.i \ \wedge \ (\forall j : anc.i.j \ \Rightarrow \ Gd.j))$ is closed for each choice of node i. (The proof for this observation is essentially the same as the proof of Theorem 3 and is left to the reader.) It follows from this observation that the $\sharp(s)$ is nonincreasing under system execution.

As long as s is not a steady state, there exists a node i such that $(\neg Gd.i \wedge (\forall j : anc.i.j \ \Rightarrow \ Gd.j))$ holds. Furthermore, as long as s satisfies $(\neg Gd.i \wedge (\forall j : anc.i.j : Gd.j))$, the fifth action of $wave.i$ is continuously enabled. Hence, by fairness, the system is guaranteed to eventually reach a state where $(Gd.i \ \wedge \ (\forall j : anc.i.j \ \Rightarrow \ Gd.j))$ holds. In other words, $\sharp(s)$ is guaranteed to eventually decrease under system execution. And thus the system is guaranteed to eventually reach a state in GD. □

From the proof of Theorem 4, it follows that if each process attempts to execute the fifth action of $wave.i$ once in every two consecutive execution attempts, then the rate of convergence of the system to a state in GD is $2 \times ht$ rounds, where ht denoted the height of the spanning tree constructed by the tree layer. Thus, the system can be implemented to ensure $O(ht)$ convergence to a state in GD.

The next two theorems imply that each distributed reset requested at a state in GD is performed correctly.

Theorem 5: Upon starting at an arbitrary state in $(GD \wedge sn.k = n)$, the system is guaranteed to reach a state in $(GD \wedge (\forall i : sn.i = n \wedge st.i \neq reset))$.

Proof: Let s be a system state in $(GD \wedge sn.k = n)$. We consider two cases:

• $\underline{((GD \wedge sn.k = n) \wedge st.k \neq reset) \text{ holds in } s}$:
From GD, the state s is seen to already satisfy $(GD \wedge (\forall i : sn.i = n \wedge st.i \neq reset))$.

• $\underline{((GD \wedge sn.k = n) \wedge st.k = reset) \text{ holds in } s}$:
The proof is by structural induction on the height of node k in the spanning tree. Note that the only action of $P.k$ which can be enabled at a state in $(st.k = reset \wedge sn.k = n)$ is its fourth action.

Base Case: (k is a leaf.)
If k is a leaf then the fourth action of process $wave.k$ is enabled at every state in $(st.k = reset \wedge sn.k = n)$. By fairness, the fourth action is eventually executed and the resulting state satisfies $(st.k \neq reset \wedge sn.k = n)$.

Induction Step: (The height of node k exceeds 0.)
Let $P.j$ be an arbitrary process such that $f.j = k$. We consider three cases:

- $\underline{(GD \wedge st.j \neq reset \wedge sn.j = n)}$:
  Since the system state satisfies $(st.k = reset \wedge sn.k = n)$, the third and fifth actions of $wave.j$ are not enabled. The second and fourth actions of $wave.j$ are not enabled either. Therefore, as long as $(st.k = reset \wedge sn.k = n)$ holds, the system state satisfies $(st.j \neq reset \wedge sn.j = n)$.
- $\underline{(GD \wedge st.j = reset \wedge sn.j = n)}$:
  Since the system state satisfies $(st.k = reset \wedge sn.k = n)$, the fourth action of $wave.j$ is the only one that can be enabled as long as the system state is in $(st.j = reset \wedge sn.j = n)$. By the inductive hypothesis, the system is guaranteed to eventually reach a state that satisfies $(st.j \neq reset \wedge sn.j = n)$, at which point the previous case applies.
- $\underline{(GD \wedge sn.j \neq n)}$:
  The third action of $wave.j$ is enabled continuously as long as $sn.j \neq n$ holds. No other enabled action of $P.j$ falsifies $sn.j \neq n$. By fairness, the third action of $wave.j$ is eventually executed, yielding a state in which one of the previous two cases applies.

Since the argument presented above holds for an arbitrary choice of j, we conclude that the system is guaranteed to reach a state in which $(\forall j \in N.k : (f.j = k) \Rightarrow (sn.k = sn.j \wedge st.j \neq reset))$ holds. The fourth action of $wave.k$ is then enabled continuously and, by fairness, it is eventually executed thereby yielding a state in $(sn.k = n \wedge st.k \neq reset)$. The previous case now applies. $\square$

Theorem 6: Upon starting from a state in $(GD \wedge (\exists i : sn.i = n \wedge st.i = initiate))$, the system is guaranteed to reach a state in $(GD \wedge (\forall i : sn.i = n+1 \wedge st.i \neq reset))$.

Proof: Let s be a system state in $(GD \wedge (\exists i : sn.i = n \wedge st.i = initiate))$. We consider two cases:

• $\underline{sn.k = n+1 \text{ holds in } s}$:
The result follows directly from Theorem 5.

• $\underline{sn.k = n \text{ holds in } s}$:
In this case, $(GD \wedge (\forall i : sn.i = n) \wedge (\exists i : st.i = initiate))$, holds at s. Due to the previous case, it suffices for us to show that the system is guaranteed to reach a state in $(GD \wedge sn.k = n+1)$.

Consider the variant function $\S(s) = \langle di(s), li(s), ln(s) \rangle$, where
$\langle di(s), li(s), ln(s) \rangle$ is a sequence of natural numbers,
$di(s) = min \; \{ \; d.i \mid st.i = initiate \}$,
$li(s) \; = K - | \; \{ \; i \mid st.i = initiate \} \; |$, and
$ln(s) = K - | \; \{ \; i \mid st.i = normal \} \; |$.
Elements in the range of $\S$ are related by the well-founded relation $\prec$ that we introduced previously.

If $di(s) > 0$ then the value assigned by $\S$ to the system state is, under system execution, *decreasing* with respect to $\prec$. To see this, note that the second, third and the fifth actions of *wave.i* cannot be enabled at s. Executing the first action or an action of type (T2) decreases $li(s)$ and does not increase $di(s)$. Finally, the fourth action preserves $di(s)$ and $li(s)$ but decreases $ln(s)$. Thus, the system is guaranteed to reach a state in which $di(s) = 0$; that is, $st.k = initiate$ holds. When $st.k = initiate$, the second action of *wave.k* is enabled and remains enabled until, by fairness, it is eventually executed to yield a state that satisfies $(GD \wedge sn.k = n+1)$. $\square$

Lastly, we analyze the time taken to complete a distributed reset. Observe that a request wave reaches the root within ht rounds. Also, a reset wave propagates from the root to the leaves within ht rounds. Since each node has to wait for messages from each of its children in a completion wave, the completion wave propagates from the leaves to the root within $min \; (dg \times ht, n)$, where dg is the maximum degree of nodes in the spanning tree and n is the number of up processes. Thus, a distributed reset initiated at any state in GD is completed within $O(\, min \, (dg \times ht, n) \,)$ rounds.

10 Appendix C

Sketch of Correctness Proof for the Low Atomicity Tree Layer.

Define $\mathcal{H}.1 \equiv$

$$\begin{aligned}
&(root.k = k \ \wedge\ f.k = k \ \wedge\ d.k = 0) \quad \wedge \\
&(\forall i:\ adj.i.k\ :\ (\tilde{root}.i.k = root.k \ \wedge\ \tilde{f}.i.k = f.k \ \wedge\ \tilde{d}.i.k = d.k) \quad \wedge \\
&(\forall i,j:\ root.i \leq k \ \wedge\ (j \in N.i \ \Rightarrow\ \tilde{root}.i.j \leq k) \ \wedge \\
&\qquad ((root.i = k \ \wedge\ dist.i.k > 0) \Rightarrow d.i > 0)) \ \wedge \\
&\qquad ((j \in N.i \ \wedge\ \tilde{root}.i.j = k \ \wedge\ dist.j.k > 0) \Rightarrow \tilde{d}.i.j > 0))
\end{aligned}$$

Lemma 5: The set of states $\mathcal{H}.1$ is closed under system execution. □

Lemma 6: Upon starting at an arbitrary state, i.e., a state in $\mathcal{H}.0$, the system is guaranteed to reach a state in $\mathcal{H}.1$. □

Define, by induction, for $l > 0$,

$$\begin{aligned}
&\mathcal{H}.(l+1) \equiv \\
&\quad \mathcal{H}.l \ \wedge \\
&\quad (\forall i : dist.i.k = l \ \Rightarrow \\
&\qquad root.i = k \ \wedge \\
&\qquad (\exists j : j \in N.i\ :\ f.i = j \ \wedge\ d.i = d.j + 1 \ \wedge\ d.j = \underline{min}\{\ d.j' \mid root.j' = k \ \wedge\ j' \in N.i\ \}) \ \wedge \\
&\qquad (\forall j:\ j \in N.i\ :\ (\tilde{root}.j.i = root.i \ \wedge\ \tilde{f}.j.i = f.i \ \wedge\ \tilde{d}.j.i = d.i))) \quad \wedge \\
&\quad (\forall i:\ (root.i = k \ \wedge\ dist.i.k > l) \ \Rightarrow\ d.i > l) \ \wedge \\
&\quad (\forall i,j:\ (adj.j.i \ \wedge\ \tilde{root}.j.i = k \ \wedge\ dist.i.k > l) \ \Rightarrow\ \tilde{d}.j.i > l)
\end{aligned}$$

Lemma 7: For each l such that $1 \leq l < K$ the following proposition holds:
$\mathcal{H}.(l+1)$ is closed under system execution. □

Lemma 8: For each l such that $1 \leq l < K$ the following proposition holds:
Upon starting at an arbitrary state in $\mathcal{H}.l$, the system is guaranteed to reach a state in $\mathcal{H}.(l+1)$. □

Proofs of Lemmata 5–8 appear in [2].

Theorem 7: {Closure of $\mathcal{G}$}
The set of states $\mathcal{G}$ is closed under system execution.
Proof: $\mathcal{G} \equiv \mathcal{H}.K$. The theorem follows from Lemma 7. □

Theorem 8: {Convergence to $\mathcal{G}$}
Upon starting at an arbitrary state, the system is guaranteed to reach a state in $\mathcal{G}$.
Proof: By transitivity, using Lemmas 6 and 8, and $\mathcal{G} \equiv \mathcal{H}.K$. □

11 Appendix D

Sketch of Correctness Proof for the Low Atomicity Wave Layer.

Define

$$
\begin{aligned}
\mathcal{G}d.i \equiv\ & ((f.i\!=\!j \ \wedge\ st.j\!\neq\! reset) \Rightarrow \\
&\qquad (st.i\!\neq\! reset \ \wedge\ sn.j\!=\!sn.i \ \wedge\ \tilde{st}.j.i\!\neq\! reset \ \wedge\ sn.i\!=\!\tilde{sn}.i.j \ \wedge\ sn.i\!=\!\tilde{sn}.j.i)) \\
& \wedge \\
& ((f.i\!=\!j \ \wedge\ st.j\!=\!reset \ \wedge\ \tilde{st}.i.j\!\neq\! reset) \Rightarrow \\
&\qquad (st.i\!\neq\! reset \ \wedge sn.j\!=\!sn.i\!+\!1 \ \wedge\ \tilde{st}.j.i\!\neq\! reset \ \wedge\ sn.i\!=\!\tilde{sn}.i.j \ \wedge\ sn.i\!=\!\tilde{sn}.j.i)) \\
& \wedge \\
& ((f.i\!=\!j \ \wedge\ st.j\!=\!reset \ \wedge\ \tilde{st}.i.j\!=\!reset) \Rightarrow \\
&\qquad ((st.i\!\neq\! reset \ \wedge sn.j\!=\!sn.i\!+\!1 \ \wedge\ \tilde{st}.j.i\!\neq\! reset \ \wedge \\
&\qquad\ (sn.i\!=\!\tilde{sn}.i.j \ \vee\ sn.i\!+\!1\!=\!\tilde{sn}.i.j) \ \wedge\ sn.i\!=\!\tilde{sn}.j.i) \\
&\qquad \vee \\
&\qquad (sn.j\!=\!sn.i \ \wedge\ \tilde{sn}.i.j\!=\!sn.i \ \wedge \\
&\qquad (\tilde{st}.j.i\!=\!reset \ \Rightarrow\ \tilde{sn}.j.i\!=\!sn.i) \ \wedge \\
&\qquad ((\tilde{st}.j.i\!\neq\! reset \ \wedge\ st.i\!=\!reset) \ \Rightarrow\ sn.i\!=\!\tilde{sn}.j.i\!+\!1) \ \wedge \\
&\qquad ((\tilde{st}.j.i\!\neq\! reset \ \wedge\ st.i\!\neq\! reset) \ \Rightarrow\ (sn.i\!=\!\tilde{sn}.j.i \ \vee\ sn.i\!=\!\tilde{sn}.j.i\!+\!1))))
\end{aligned}
$$

and

$\mathcal{GD} \ \equiv\ (\forall i : \mathcal{G}d.i)$

Theorem 9: {Closure of $\mathcal{GD}$}
The set of states $\mathcal{GD}$ is closed under the execution of the system. □

Theorem 10: {Convergence to $\mathcal{GD}$}
Upon starting at an arbitrary state, the system is guaranteed to reach a state in $\mathcal{GD}$. □

Theorem 11: Upon starting at an arbitrary state in $(\mathcal{GD} \ \wedge\ sn.k\!=\!n)$, the system is guaranteed to reach a state in $(\mathcal{GD} \ \wedge\ (\forall i : sn.i\!=\!n \ \wedge\ st.i\!\neq\! reset))$. □

Theorem 12: Upon starting from a state in $(\mathcal{GD} \ \wedge\ (\exists i : sn.i = n \ \wedge\ st.i = initiate))$, the system is guaranteed to reach a state in $(\mathcal{GD} \ \wedge\ (\forall i : sn.i\!=\!n\!+\!1 \ \wedge\ st.i\!\neq\! reset))$. □

Proofs of Theorems 9–12 appear in [2].