Once-and-Forall Management Protocol (OFMP)

Sandeep S. Kulkarni *Anish Arora*

Department of Computer and Information Science *
The Ohio State University
Columbus, OH 43210 USA

Abstract

OFMP is a hierarchical, network management protocol that enables group operations to be executed on the management information bases of all nodes in the group. As long as no faults occur, OFMP ensures that all nodes execute their local operation exactly once in each group operation. If "immediately-detectable" faults occur, it ensures masking fault-tolerance; i.e., all non-failed nodes execute their local operation exactly once in each group operation. And, if "eventually-detectable" faults occur, it ensures stabilizing fault-tolerance; i.e., it eventually converges to a state from where all non-failed nodes execute their local operation exactly once in each subsequent group operation. Of special note is the ability of OFMP to detect using only a bounded amount of memory whether nodes have executed in same group operation, in a manner that masks immediately-detectable faults and stabilizes from eventually-detectable faults.

Keywords : network management, group operations, robust hierarchical control, diffusing computations, fault-tolerance, bounded memory

1 Introduction

Extant network management protocols such as SNMP and CMIP $[1-3]$ provide mechanisms for a management client process to invoke an operation at a management server process. Typically, the operation is some sort of fetch or store on an object that is maintained locally at the server in its management information base (MIB). In some cases, the operation has side-effects on the resources maintained by the server; it may also return a response to the client upon its completion or when some event occurs subsequently.

While the client-server architecture of network control is indeed simple and it is wellsuited to heterogeneous networks, it is less than ideal for several types of control domains. Examples of such control domains include those that contain an inherently hierarchical organization of servers, that contain proxy servers or anonymous (but indirectly reachable) servers, or that require concurrent "group" operations on the MIBs at multiple servers.

With these more general control domains in mind, we consider in this paper a hierarchical architecture of network control as an alternative to the client-server architecture. Specifically, we present a hierarchical protocol which we refer to as the "once-and-forall management protocol" (OFMP). Informally, the specification of OFMP enables concurrent execution of operations so that (1) each operation is executed on the MIB of some server with *exactly-once* semantics, and (2) an operation is executed at *all* servers in the control domain.

The main source of difficulty in OFMP is in dealing with the occurrence of faults. We are interested in considering a variety of faults, such as communication faults, memory faults, processing faults, and security faults. While the details of the faults are relegated to the next section, we may informally classify these faults into two: faults in the presence of which (1) and (2) can always be satisfied and faults in the presence of which (1) and (2) cannot always be satisfied. OFMP is designed to "mask" the occurrence of faults in the first fault-class. And, although it cannot possibly mask the occurrence of the second fault-class, OFMP is designed to "stabilize" from the occurrence of the second fault-class, in the following sense. Even if (1) and (2) are violated for some invocations of OFMP due to occurrence of faults in second fault-class, continued execution of OFMP ensures that eventually (1) and (2) are satisfied for all subsequent OFMP invocations.

A secondary source of difficulty is bounding the memory of OFMP processes and, hence, the size of OFMP messages. Bounding the memory raises the issue of how to detect whether multiple processes have participated in the same OFMP invocation. (Keeping a bounded sequence number does not suffice by itself since sequence numbers for old OFMP invocations may remain in the network. This may occur because some communication channels may be slower than others; communication channels may allow messages to be reordered; processes may repair with incorrect sequence numbers; or transient state faults may arbitrarily corrupt the sequence numbers. In Section 4, we give an example to illustrate that two processes may have the same sequence number even though they have participated in different OFMP invocations.) This detection is hard even if we only consider how to stabilize from the second fault-class; it becomes even harder if only consider how to mask the first fault-class; and it is especially hard if we consider both how to mask and how to stabilize (cf. Sections 4 and 5). In fact, we are not aware of any work by others on group operations that has yielded bounded memory protocols that are masking as well as stabilizing.

---

[0]Email:{kulkarni,anish}@cis.ohio-state.edu;
Web: http://www.cis.ohio-state.edu/{~kulkarni,~anish}.
Research supported in part by NSF Grant CCR-93-08640, NSA Grant MDA904-96-1-1011, and OSU Grant 221506

The rest of this paper is organized as follows. In Section 2, we give the requirements of OFMP, in the absence of faults and in the presence of faults in each of the two fault-classes. In Section 3, we note that in the absence of faults the requirements of OFMP are simply met by performing a diffusing computation. We then present, in Section 4, a bounded-memory extension of diffusing computations that solves the problems encountered in the masking the first fault-class. We show that OFMP is stabilizing from the second fault-class, in Section 5. Finally, we discuss extensions of OFMP in Section 6 and make concluding remarks in Section 7.

2 Problem Statement

Given is a connected network with bidirectional channels. One node of the network has the unique identifier ROOT associated with it. (The other network nodes may or may not have identifiers associated with them.) Only the ROOT node can initiate an OFMP computation. In the absence of faults, each OFMP computation is required to satisfy the following specification.

- *Exactly-Once* : Every node executes some local operation exactly once in that OFMP computation.
- *Safety* : ROOT completes that OFMP computation only after all nodes have executed their local operations.
- *Progress* : ROOT completes that OFMP computation eventually.

Immediately-detectable faults. Some of the faults that occur in the network can be detected by one or more nodes without interfering with any step of the protocol. In other words, the interval from the occurrence of these faults to their detection can be viewed as being "atomic" with respect to the individual steps of the protocol. For example, a message loss and its detection can be viewed as atomic if we can "pretend" as if the fault occured just when the process detected it. Thus, these faults can be viewed as being "immediately-detectable" and, for reasoning about the fault-tolerance of the protocol, the occurrence of the fault and the corresponding detection can both be modeled in a single "fault action".

The immediately-detectable faults we consider include node fail-stops, node repairs, channel fail-stops, and channel repairs. For ease of exposition, we assume that these immediately-detectable faults do not partition the network, nor do they fail-stop the node ROOT. (We will later discuss, in Section 7, how these two assumptions are relaxed.)

In the presence of these immediately-detectable faults, it is not possible to ensure that all nodes complete their local operation. Therefore, in the presence of immediately-detectable faults, each OFMP computation is required to satisfy a weakened version of the specification above that is obtained by replacing the Exactly-Once and Safety conditions with the following two conditions:

- *Atmost-Once* : Every node that does not fail during that computation executes some local operation exactly once in the OFMP computation.
- *Weak-Safety* : ROOT completes that OFMP computation only after all nodes that do not fail during the computation have executed their local operations.

Eventually-detectable faults. The remaining faults that the network is subject to cannot be immediately detected by the nodes, but they can be detected eventually. Since these "eventually-detectable" faults are not atomic, they can interfere with individual protocols steps and cause them to execute incorrectly.

The eventually-detectable faults we consider include transient state corruption (whereby protocol steps may access incorrect state and yield incorrect executions) and node crashes where incorrect state is output before the crash (whereby other nodes may execute protocol steps incorrectly before they detect the crash).

In the presence of eventually-detectable faults, since some nodes may execute protocol steps incorrectly during an OFMP computation, it is possible that some OFMP computations do not satisfy even the weakened specification given above. Therefore, in the presence of a finite number of eventually-detectable faults, it is required that eventually the network resumes the satisfaction of the weakened specification for all subsequent OFMP computations. In other words, the network is required to be self-stabilizing with respect to eventually-detectable faults.

Protocol notation. OFMP is formally defined by a set of *variables* and a finite set of *actions* at each node. Each variable ranges over a predefined nonempty domain, and each action is of the form:

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

The guard of each action is a boolean expression over the protocol variables. The execution of the statement of each action atomically and instantaneously updates zero or more protocol variables.

A *state* of OFMP is defined by a value for each variable of OFMP, chosen from the domain of the variable. A state predicate of OFMP is a boolean expression over the variables of OFMP. An action of OFMP is *enabled* in a state iff its guard (state predicate) evaluates to true in that state.

A *computation* of OFMP is a fair, maximal sequence of steps; in every step, an action of OFMP that is enabled in the current state is chosen and the statement of the action is executed atomically. Fairness of the sequence means that each action in OFMP that is continuously enabled along the states in the sequence is eventually chosen for execution. Maximality of the sequence means that if the sequence is finite then the guard of each action in OFMP is false in the final state, i.e., the computation continues until a state is reached in which no action is enabled.

3 A Simple Fault-Intolerant Version of OFMP

In the absence of faults, a diffusing computation suffices to meet the requirements of OFMP. To implement the diffusing computation, we assume the existence of the following underlying network service.

Assumption. There is a service that maintains a logical tree spanning the nodes and rooted at ROOT. The service notifies each node, j, of its parent, $par.j$, in the tree. (For simplicity, par.ROOT =ROOT.) Thus, in the absence of faults, each j has a path to ROOT via $par.j$. Since the tree can be corrupted in the presence of faults, for instance when not all nodes or channels remain up, the service guarantees that it eventually changes the parent relation so as to reconfigure the tree on the remaining up nodes. In the interim, j need not have a path to ROOT via $par.j$ nor does it know this fact.

ROOT initiates a diffusing computation as follows. It executes its own local operation and propagates the diffusing computation to its children. The children in turn execute their local operation and further propagate the diffusing computation to their children. This continues until the diffusing computation reaches the leaf nodes. Subsequently, each node completes in the diffusing computation, but only when all its children have completed in the diffusing computation; hence, the leaf nodes complete first, then their parents, and so on. It follows that when ROOT completes the diffusing computation, all nodes in the network have completed the diffusing computation and executed their local operation exactly once. Thus, the requirements of OFMP in the absence of faults are met.

We program the diffusing computation by letting j maintain a state, $st.j$, whose value is either $prop$ or $comp$, to denote whether j is propagating a diffusing computation or has completed a diffusing computation, and a sequence number, $sn.j$, whose value is either 0 or 1, to distinguish between successive diffusing computations. Node j has three actions, which are given below. The first is the initiation action (which can execute only if j =ROOT), the second is the propagation action, and the third is the completion action (let $par.j$ denote the parent of j and $ch.j$ denote the children of j):

j =ROOT $\wedge$ $st.j = comp$ $\wedge$
{j needs to initiate a diffusing computation }
$\longrightarrow$ $st.j,\ sn.j := prop,\ sn.j \oplus 1$; { execute local operation at j }

$st.(par.j) = prop$ $\wedge$ $sn.j \neq sn.(par.j)$ $\wedge$ $st.j = comp$
$\longrightarrow$ $st.j,\ sn.j := prop,\ sn.(par.j)$; { execute local operation at j }

$st.j = prop$ $\wedge$ $(\forall k : k \in ch.j : sn.j = sn.k \ \wedge\ st.k \neq prop)$
$\longrightarrow$ $st.j := comp$; { if j = ROOT then declare OFMP complete }

Proof of correctness. Observe that if both j and $par.j$ are propagating a diffusing computation then they have the same sequence number. Also, if $par.j$ has completed a diffusing computation then j has also completed that diffusing computation. Hence, the predicate $GD = (\forall j :: Gd.j)$ is invariantly true at all states in every OFMP computation, where

$$Gd.j = ((st.(par.j) = prop \ \wedge\ st.j = prop) \ \Rightarrow\ sn.j = sn.(par.j)) \ \wedge$$
$$((st.(par.j) \neq prop) \ \Rightarrow\ (st.j \neq prop \ \wedge\ sn.j = sn.(par.j)))$$

Lemma 1 Starting from any state where GD is satisfied and the graph of the parent relation forms a rooted tree, every computation of OFMP satisfies *Exactly-Once, Safety*, and *Progress*.

4 Masking Immediately-Detectable Faults

In this section, we show how OFMP is extended so that it satisfies *Atmost-Once, Weak-Safety*, and *Progress* in the presence of immediately-detectable faults. We begin by identifying the difficulties involved in dealing with immediately-detectable faults.

One difficulty is that if a node fail-stops before completing a diffusing computation, its parent will never complete that diffusing computation, causing the diffusing computation to deadlock and thereby violating *Progress*. A second difficulty is how to preserve *Atmost-Once* if OFMP repeats a diffusing computation in order to deal with a previous diffusing computation that could not reach all nodes. A third difficulty is preserving *Weak-Safety* if a diffusing computation is initiated in a state where the tree is partial, i.e., does not span all up nodes. Finally, the most severe difficulty is how to determine using only bounded memory whether a node has propagated the current diffusing computation of ROOT. We show how to deal with these difficulties, next.

Dealing with fail-stopped nodes and channels. When a node detects that its neighbor or the channel between them has fail-stopped, it "aborts" any diffusing computation it is propagating by completing that diffusing computation with a "result" of false. This allows the parent of that node to likewise complete that diffusing computation with the result false, and so on, until ROOT completes the diffusing computation with the result false. If ROOT completes a diffusing computation with the result false, it starts a new diffusing computation. And, if ROOT completes with the result true, ROOT completes that OFMP computation. Thus, both *Progress* and *Weak-Safety* are preserved.

Preserving Atmost-Once. Since ROOT may require multiple diffusing computations to reach all nodes if immediately-detectable faults occur, to preserve *Atmost-Once*, an operation number, *opn*, is associated with each OFMP computation. When a node executes its local operation, it remembers the operation number. By including the operation number in each diffusing computation, it is possible for each node to detect whether it has previously executed a local operation with that operation number.

Clearly, a minimum of two values (say 0 and 1) are necessary for *opn*. But, as shown in the following example (see Figure 1), two values of *opn* are insufficient to preserve *Atmost-Once*. To see this, consider two computations: In the first computation, the initial state is as shown in Figure 1(a). ROOT initiates a diffusing computation, nodes l and j propagate this diffusing computation, and execute their local operation (see Figure 1(b)). Now, node l fails and the tree is reorganized as shown in Figure 1(c). In the second computation, the initial state is as shown in Figure 1(d). In this state, ROOT propagates a diffusing computation (see Figure 1(e)). Observe that in Figures 1(c) and (e), the view of node j is identical. In the first case, j should not execute its local operation, whereas in the second case, it should. Thus, keeping only two values for the operation number is insufficient for a node to determine the current operation number used by ROOT.

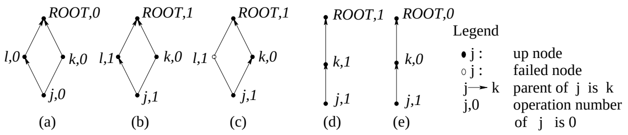

Figure 1: Necessity of three operation numbers

If the operation number is either 0, 1 or 2, and we follow the rule that operation number $B+1$ is used only if the operation number $B-1$ does not exist in the network, in any state only two operation numbers coexist. (In this sequel, we let $+$ and $-$ denote modulo 3 addition and subtraction respectively.) And, for a pair of operation numbers, B and $B+1$, $B+1$ is the operation number used by ROOT. Thus, when j propagates a diffusing computation, j executes its local operation iff $opn.(par.j)$ is one greater than $opn.j$.

Dealing with diffusing computations initiated on a partial tree. Recall that if the tree is partial, the underlying network service eventually reconfigures the tree. To ensure that a diffusing computation completes correctly after the tree is reconfigured, we add an action at node j that ensures that eventually $Gd.j$ is satisfied.

Moreover, to ensure that an OFMP computation does not complete unsafely until the tree reconfigures, we exploit the fact that the nodes remain connected in the presence of faults. It follows that if some nodes have not propagated a diffusing computation then there exists at least one pair of neighboring nodes, say j and l, such that j propagated the diffusing computation but l did not. To detect the existence of such pairs, each node detects whether all its neighbors have propagated the current diffusing computation: node j completes a diffusing computation with the result true only if all its neighbors have propagated the diffusing computation and, hence, executed their local operation. We address this issue of how to detect whether two nodes have propagated the same diffusing computation, next.

Detecting whether a neighbor has propagated the same diffusing computation. As noted previously, this is the most challenging part in the design of OFMP. Let us begin by exhibiting a scenario to demonstrate that: (i) it is possible that two nodes, j and l, have the same sequence number even if j has propagated the current diffusing computation and l has not, and (ii) it is possible that even if j completes one diffusing computation after it changes its parent and j and l have the same sequence number, j has propagated the current diffusing computation and l has not. Consider the computation starting from the state in Figure 2 (a). In this state, ROOT has initiated a diffusing computation with sequence number 0, and all nodes except l have propagated that diffusing computation. The computation proceeds as follows:

- Node n fails.
- ROOT completes its diffusing computation and initiates a new one with sequence number 1 (Figure 2(b)).
- Node j separates from the tree, changes its parent to ROOT, and propagates the diffusing computation with sequence number 1 (Figure 2(c)).
- Node j completes its diffusing computation (Figure 2(d)). Observe that when j completes this diffusing computation, although the sequence number of l is the same as that of j, l has not propagated the current diffusing computation of ROOT. Thus, (i) is satisfied.
- Node l propagates the diffusing computation with sequence number 0. Also, ROOT completes its diffusing computation and initiates a new one with sequence number 0 (Figure 2(e)).
- Node j propagates the new diffusing computation of ROOT with sequence number 0 (Figure 2(f)). Observe that although j has completed one diffusing since it changed its parent and the sequence numbers of j and l are the same, they are in different diffusing computations. Thus, (ii) is satisfied.

From the above scenario, it follows that after j changes its parent, for two subsequent diffusing computations, j cannot safely detect whether l has propagated the current diffusing computation. To see that j can safely detect this in the third diffusing computation, we argue as follows.

Consider the set, $anc.l$, of ancestors of l that satisfy the predicate Gd. If there exist two nodes m and n in $anc.l$ such that $par.m = n \;\wedge\; sn.m \neq sn.n$ then, from $Gd.m$, $st.m$ is equal to *comp* and $st.n$ is equal to *prop*. Moreover, all ancestors of n have the same sequence number as n, and all descendents of m have the same sequence number as m (see Figure 3). Since l can obtain a different sequence number only from its ancestors, we observe that as long as the set $anc.l$ does

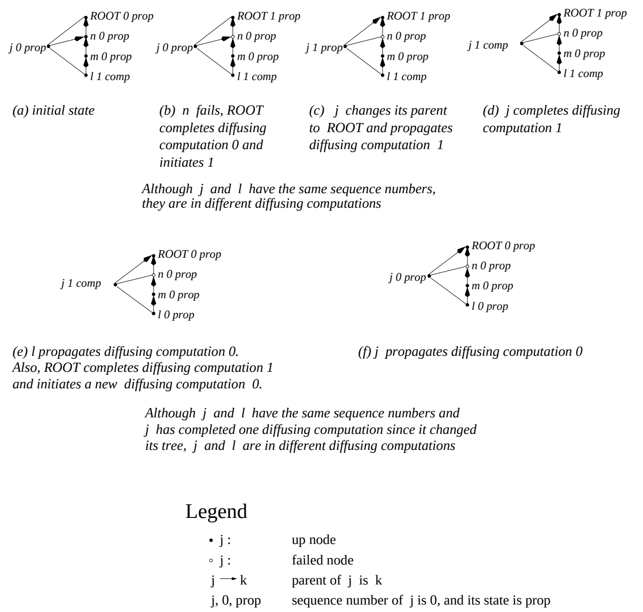

Figure 2: Detecting whether a neighbor has propagated the current diffusing computation

not increase, l can propagate at most two old diffusing computations. Moreover, we observe that each time $anc.l$ increases, l may be able to propagate at most two more old diffusing computations.

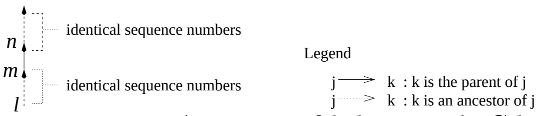

Figure 3: Ancestors of l that satisfy Gd

These observations allow j to safely detect whether its neighbor l has propagated the current diffusing computation, as follows: When j completes a diffusing computation, the sequence numbers of j and l are identical. It follows that when j completes two diffusing computations, l must also have propagated two diffusing computations (albeit possibly different ones), *assuming* $anc.l$ has not increased in the meanwhile. To ensure that j completes with a result of false in both of these diffusing computations and with a result of true in the next diffusing computation, we let j maintain a ternary variable $rc.j$ (for retry count) whose value is either 0, 1 or 2. When j changes its parent, $rc.j$ is set to 2. When j completes a diffusing computation, if $rc.j$ is nonzero, j decrements it by 1. Thus, j detects that it has completed at least two diffusing computations since it last changed its parent by checking that $rc.j$ is zero. And, only when $rc.j$ is zero does j complete the diffusing computation with a result of true.

For the case where $anc.l$ does increase in the meanwhile, we observe that $anc.l$ increases only when some ancestor of l changes its parent or satisfies Gd. In this case, l may propagate at most two more old diffusing computations. We ensure that whenever l propagates the first of these diffusing computations l can detect that $anc.l$ may have increased. To this end, whenever a node satisfies Gd, we let it set its rc value to 2. We also let a node set its rc value to 2 upon propagation of a diffusing computation if the rc value of its parent is 2. It follows that if $anc.l$ increases, then in the next diffusing computation that l propagates, $rc.l$ is set to 2. Furthermore, if $rc.l$ is zero then l has propagated its old diffusing computations, and the next diffusing computation it propagates is the current diffusing computation of ROOT.

Theorem 2 (**A sufficient condition for detecting that nodes are in the same diffusing computation**) Let j and l be neighboring nodes and let ROOT be an ancestor of j. In every OFMP computation if $rc.j$ and $rc.l$ are both zero and $sn.j = sn.l$ holds then l has propagated the same diffusing computation as j.

Variables and actions of OFMP. Based on the solution described above, two ternary variables $opn.j$ and $rc.j$ are added at each node j. The actions of j are obtained by modifying the actions of the intolerant version of OFMP, as follows.

The initiation action is modified so that when ROOT initiates a diffusing computation, it increments opn.ROOT.

The propagation action is modified so that when j propagates a diffusing computation, it executes its local operation iff the operation number of $par.j$ is one greater than that of j. Also, if $rc.(par.j)$ is equal to 2, i.e., if $par.j$ has *recently* changed parent or satisfied $Gd.(par.j)$, j sets $rc.j$ to 2. (By *recently*, we mean that $par.j$ has not executed the completion action since it changed parent or satisfied $Gd.(par.j)$.)

The completion action is modified so that j completes a diffusing computation only when the sequence numbers of all its neighbors are the same as that of j. Also, j checks whether the rc values of j and its neighbors are equal to zero. If the rc value of j or one of its neighbors is non-zero, j aborts this diffusing computation, by setting $rc.(par.j)$ to a non-zero value. This ensures that whenever $par.j$ completes its diffusing computation, $par.j$ aborts that diffusing computation, and so on. Thus, when ROOT completes the diffusing computation, its rc value is non-zero and, hence, it concludes that the diffusing computation completed incorrectly. If ROOT completes a diffusing computation incorrectly, it starts a new one. And, if ROOT completes a diffusing computation correctly, it completes OFMP. (Note that the "result" of a diffusing computation is thus implemented using rc; the result is true iff rc value is zero.)

Moreover, two actions are added at j. The first of these satisfies $Gd.j$ by setting $sn.j = sn.(par.j)$ and $st.j = comp$, and also sets $rc.j$ to 2. The second is superposed on the underlying tree correction protocol. Whenever the underlying tree correction protocol changes the parent of j, j sets $rc.j$ to 2. Also, since the previous parent of j cannot get a completion of diffusing computation from j, the previous parent of j aborts its diffusing computation by setting its rc value to 2.

Formally, the actions of OFMP are as shown in Figure 4. In the figure, $Adj.j$ denotes the set of neighbors of j that are up. It is implicit that the actions of j are executed only if j is up.

Actions for immediately-detectable faults. As discussed in Section 2, for reasoning about the fault-tolerance of OFMP, the occurrence of an immediately deteable fault and its corresponding detection can both be modeled by a single fault action. For example, when j fail-stops or repairs, the execution of the actions of the up neighbors of j is not affected by how long it takes to detect the fault. Hence, the fail-stop and its detection can both be modeled by one fault action. (By contrast, if neighboring up nodes access corrupted state after j crashes, the execution of their actions is affected by how long it takes to detect that crash; hence, the eventually-detectable crash fault and its corresponding detection cannot be modeled by one action.)

When j fail-stops, the neighbors of j set their rc values to 2, since they cannot detect whether j has propagated the current diffusing computation. Likewise, when the channel between j and k fail-stops, both j and k set their rc value to 2.

When j repairs, j sets its *st* value to *comp*, since it is no longer obliged to be propagating any diffusing computation. Also, it sets its rc value to 2, since j may no longer have a path (in the parent relation) to ROOT and it may end up propagating two old diffusing computations. Finally, it copies the operation number of one of its neighbors, since its operation number may no longer be valid. Likewise, when the channel between j and k repairs, at least one of them, say j, sets $rc.j$ to 2 and $st.j$ to *comp*. (Note that $opn.j$ is still valid.) Formally, these fault actions are as shown in Figure 4.

Theorem 3 Starting from an initial state where the operation numbers of all processes are the same and their retry counts are 2, every computation of OFMP in the presence of a finite number of immediately-detectable faults satisfies *Atmost-Once, Weak-Safety*, and *Progress*.

Proof of correctness. The proof is based upon characterizing the set of states reached by OFMP in the presence of immedicately-detectable faults. For reasons of space, we have relegated it to the technical report version of the paper [4].

5 Stabilizing from Eventually-Detectable Faults

OFMP satisfies *Atmost-Once, Weak-Safety* and *Progress* in the presence of immediately-detectable faults, but not in the presence of eventually-detectable faults. For instance, if a node crashes and its neighbors receive inconsistent state information before they receive notification of the failure, or if eventually-detectable message corruption, memory corruption or topology information corruption occur, OFMP may reach a state from where its specification is violated.

For reasoning about eventually-detectable faults, we model them as arbitrary changes to the state of OFMP and the set of up nodes and channels.

In this section, we show that after the occurrence of any finite number of eventually-detectable faults, OFMP eventually recovers to a state from where *Atmost-Once, Weak-Safety* and *Progress* are satisfied. This recovery occurs in five steps: First, the parent tree is reconfigured. Then, GD is satisfied. Next, ROOT completes its current diffusing computation. Subsequently, the rc values of all nodes become zero. Finally, the *opn* values of all nodes are synchronize. We address each of these steps, next.

Stabilization of the parent tree. Starting from an arbitrary state, the underlying tree correction service reconfigures the parent tree to span all up nodes. (We note that several stabilizing tree correction services have been discussed in literature, any one of which would suffice for our purposes.)

Stabilization of GD. $Gd.ROOT$ is always trivially satisfied. Once the parent tree is reconfigured, for any child of ROOT, say j, if $Gd.j$ is not satisfied, the Gd-correction action is enabled at j. Hence, $Gd.j$ is eventually satisfied. It follows that eventually all children of ROOT satisfy Gd. By induction on the depth of the tree, eventually all nodes satisfy Gd; hence, GD is eventually satisfied.

Stabilization of the diffusing computations. After the parent tree is reconfigured and GD is satisfied, the tree-correction and Gd-correction actions do not execute. Now, consider the variant function $|\{j : st.j = prop\}| + 2|\{j : sn.j \neq sn.ROOT\}|$. When a node executes either the propagation or the completion action, the value of this function decreases. Thus, if only propagation and completion actions execute, eventually the network will reach a state where the value of this function is zero, in which case the state of all nodes is *comp* and their sequence numbers are the same as $sn.ROOT$. Also, if the initiation action is ever executed then, from GD, the state of all nodes is *comp* and their sequence numbers are the same as $sn.ROOT$. It follows that starting from any state, OFMP reaches a state where the state of all nodes is *comp* and the sequence numbers of all nodes are identical. Subsequent diffusing computations are guaranteed to propagate to all nodes and to complete.

Stabilization of rc. Since every diffusing computation completes, after two subsequent diffusing computations, the rc values of all nodes will be zero.

Stabilization of *opn*. Let $opn.ROOT = B$ upon completion of a full diffusing computation after the first four steps have occurred. In this case, the *opn* of the children of ROOT is either B or $B+1$. (Had their *opn* values been $B-1$ prior this diffusing computation, they

Constants

j, k, l : node

Variables

$st.j$: $\{prop, comp\}$
$sn.j$: $\{0, 1\}$
$opn.j, rc.j$: $\{0, 1, 2\}$
$par.j$: node

Actions

Initiation action::

$j = ROOT \;\wedge\; st.j = comp \;\wedge$ $\{j$ needs to initiate a diff. comp.$\}$ $\longrightarrow$ $st.j,\ sn.j, opn.j := prop,\ sn.j \oplus 1, opn.j + 1$; $\{$ execute local operation at j $\}$

Propagation action::

$st(par.j) = prop \wedge$
$sn.j \neq sn.(par.j) \;\wedge$
$st.j = comp$
$\longrightarrow$ $st.j,\ sn.j := prop,\ sn.(par.j)$;
 if $(1 + opn.j = opn.(par.j))$ then
  $opn.j := opn.(par.j)$, $\{$ execute local operation at j $\}$;
 if $rc.(par.j) = 2$ then $rc.j := 2$

Completion action::

$st.j = prop \wedge$
$(\forall l : l \in Adj.j : sn.l = sn.j) \wedge$
$(\forall l : l \in ch.j : st.l \neq prop)$
$\longrightarrow$ $st.j := comp$;
 if $(\exists\, l :\; l \in Adj.j \cup \{j\} : rc.l > 0)$ then
  if $(j = ROOT)$ then $st.j,\ sn.j := prop, sn.j \oplus 1$
  else $rc.(par.j) := max(rc.(par.j), 1)$;
 else if $(j = ROOT)$ then $\{$ declare OFMP complete $\}$;
 $rc.j := max(0, rc.j - 1)$

Gd-correction action::

$\neg Gd.j$ $\longrightarrow$ $st.j, sn.j, rc.j := comp, sn.(par.j), 2$

Tree-correction action::

$\{$ j needs to change $par.j$ to k $\}$ $\longrightarrow$ $rc.j, rc.(par.j) := 2, 2$; $par.j := k$

Fault-Actions

$\{$ j fail-stops $\}$ $\longrightarrow$ $(\forall\, l :\; j \in Adj.l : rc.l := 2)$
$\{$ channel $\langle j, k\rangle$ fail-stops $\}$ $\longrightarrow$ $rc.j, rc.k := 2, 2$
$\{$ j repairs $\}$ $\longrightarrow$ $rc.j, st.j, opn.j := 2, comp, (any\ k : k \in Adj.j : opn.k)$
$\{$ channel $\langle j, k\rangle$ repairs $\}$ $\longrightarrow$ $rc.j, st.j := 2, comp$

Figure 4: OFMP and Its Immediately-Detectable Faults

would have set *opn* to B when they propagated this computation.) Hence, when ROOT initiates the next diffusing computation and increments $opn.ROOT$, the children of ROOT will have *opn* values of $B+1$ upon propagation, i.e., their *opn* will be the same as that of ROOT. By induction on the depth of the tree, the *opn* values of all nodes will eventually be the same.

From these five steps, we get:

Theorem 4 Starting from an arbitrary state of OFMP, every computation of OFMP in the presence of a finite number of eventually-detectable faults reaches a state from where it satisfies *Atmost-Once, Weak Safety* and *Progress.*

6 Extensions of OFMP

Refinement to low atomicity. The completion and the tree-correction actions of OFMP have high atomicity in the sense that they allow a node j to update its state and the state of its parent atomically. These actions can be refined into lower atomicity actions because the parent of j completes its diffusing computation only after it receives a response from j. It follows that if j changes its tree or is subject to an immediately detectable fault, $par.j$ cannot complete its diffusing computation until it detects this. Therefore, it suffices to set $rc.(par.j)$ to 2 whenever $par.j$ detects this. This refinement and the refinement of the immediately detectable fault is discussed in [4].

Dealing with the failure of ROOT. Our assumption that ROOT does not fail can be relaxed. Clearly, if ROOT fails during an OFMP computation, *Progress* of that computation may be violated. However, some other node can be chosen in a fault-tolerant manner to initiate future OFMP computations. Such a node may be chosen using a primary-backup protocol, a robust leader election protocol or, as shown in [5], by appropriate choice of the underlying tree service. After the underlying tree service reconfigures a tree rooted at at the newly chosen node every OFMP computation will satisfy *Atmost-Once, Weak-Safety* and *Progress*.

With this extension, if the network becomes partitioned, the OFMP specification will continue to be satisfied with respect to each partition. On a related note, to allow multiple nodes to initiate OFMP, a tree can be configured at each initiator permitting multiple OFMP invocations to complete independently.

Ensuring *Safety* in the presence of immediately detectable faults. As described in Sections 2 and 4, if some node fails during OFMP computation and, hence, does not execute its local operation, *Safety* and *Progress* cannot be satisfied simultaneously. Our design of OFMP chose to guarantee *Progress* by satisfying a weaker notion of *Safety*, namely *Weak-Safety*.

In some applications, *Safety* is more important than *Progress*; for example, in the case where the root wishes to ensure that each node executes its local operation, say for reasons of non-repudiation. OFMP can be modified for such applications as follows: In the completion of the diffusing computation, let each node j compute the number of descendents that propagated the diffusing computation *via* j. This count is propagated towards ROOT in the completion of the diffusing computation. It follows that when ROOT completes its diffusing computation successfully, ROOT can detect whether all nodes in the network propagated that diffusing computation. (Note that this extension additionally assumes that ROOT knows all the nodes that should execute the operation and, hence, it cannot be used in control domains with anonymous servers or proxy servers.)

Coordinating MIB operations. Other network applications may access the MIBs while the MIBs are being updated in an OFMP computation. In some of these applications, it may be necessary that a consistent view is obtained from the accessed MIBs, in the sense that the timestamp or the version number of all accessed objects is identical; either the one prior to the OFMP or the one after the OFMP.

The operation number *opn* can be used to detect whether the accessed MIBs are in a consistent state. Whenever an application reads the MIB at node j, it also reads the operation number of j. It follows that if the application obtains objects from different MIBs that have the same operation number, the corresponding state forms a consistent state.

7 Concluding Remarks

In this paper, we presented a hierarchical, network management protocol, OFMP, that implements group operations on MIBs of all nodes in the group. OFMP is multitolerant in that it offers two fault-tolerance properties: it masks a class of immediately-detectable faults, and stabilizes from a class of eventually-detectable faults. Although, various examples of multitolerant programs are presented in the literature $[6-9]$, to our knowledge, OFMP is first multitolerant bounded memory protocol for group operations that masks immediately-detectable faults and stabilizes from eventually-detectable faults. In particular, OFMP is the first protocol that guarantees *Weak-Safety* and *Atmost-Once* in the presence of immediately-detectable faults.

OFMP was designed using diffusing computations. As discussed in Section 4, the main difficulty in the design of OFMP was detecting whether a node had last propagated the current diffusing computation. We showed that after a node changes its parent, it could not detect in the next two diffusing computations whether its neighbor has propagated the current diffusing computation. But, it could detect this in the third diffusing computation.

OFMP makes reasonable assumptions about the network: It merely assumes a tree correction service that eventually configures the nodes in the network in a tree; it makes no assumption about how the tree correction service behaves until it the tree is configured. Also, as discussed in Section 6, our assumptions that the initiator of OFMP is unique, the initiator does not fail, and the network remains connected can all be relaxed. Finally, we note that OFMP can be systematically transformed so that nodes do not access the state of their neighbors directly but only via asynchronous message passing.

References

[1] W. Stallings. *SNMP SNMPv2 and CMIP: The practical guide to network management standards.* Addison-Wesley Publishing Company, 1993.

[2] A. S. Tanenbaum. *Computer networks.* Prentice Hall, 1996.

[3] D. E. Comer. *Internetworking with TCP/IP Vol 1*, chapter 26. Prentice Hall, 1995.

[4] S. S. Kulkarni and A. Arora. Once-and-forall management protocol (OFMP). Technical Report OSU-CISRC-5/97-TR29, Ohio State University, 1997.

[5] S. S. Kulkarni and A. Arora. Multitolerance in distributed reset. Technical Report *OSU-CISRC TR13*, Ohio State University, 1996.

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

[7] A. Gopal and K. Perry. Unifying self-stabilization and fault-tolerance. *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, pages 195–206, 1993.

[8] I. Yen and F. Bastani. A highly safe self-stabilizing mutual exclusion algorithm. *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 1995.

[9] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 1995.