Auditable Restoration of Distributed Programs∗

Reza Hajisheykhi, Mohammad Roohitavaf, Sandeep S. Kulkarni
Computer Science and Engineering Department
Michigan State University
East Lansing, MI, USA
Email: {hajishey, roohitav, sandeep}@cse.msu.edu

Abstract—We focus on a protocol for auditable restoration
doing distributed systems. The need for such protocol arises due
to conflicting requirements (e.g., access to the system should be
restricted but emergency access should be provided). One can
design such systems with a tamper detection approach (based on
the intuition of break the glass door). However, in a distributed
system, such tampering, which are denoted as auditable events,
is visible only for a single node. This is unacceptable since the
actions they take in these situations can be different than those in
the normal mode. Moreover, eventually, the auditable event needs
to be cleared so that system resumes the normal operation.

With this motivation, in this paper, we present a protocol for
auditable restoration, where any process can potentially identify
an auditable event. Whenever a new auditable event occurs, the
system must reach an auditable state where every process is aware
of the auditable event. Only after the system reaches an auditable
state, it can begin the operation of restoration. Although any
process can observe an auditable event, we require that only
authorized processes can begin the task of restoration. Moreover,
these processes can begin the restoration only when the system
is in an auditable state. Our protocol is self-stabilizing and can
effectively handle the case where faults or auditable events occur
during the restoration protocol. Moreover, it can be used to
provide auditable restoration to other distributed protocol.

Keywords—Self-stabilization, reactive systems, adversary, formal
methods.

I. INTRODUCTION

A. A Brief History and the Need for Auditable Restoration

Fault-tolerance focuses on the problem of what happens if
the program is perturbed by undesired perturbations. In other
words, it focuses on what happens if the program is perturbed
beyond its legitimate states (a.k.a. invariant). There have
been substantial ad-hoc operational approaches designed for
specific types of faults for providing fault-tolerance. For
example, the idea of recovery blocks [16] introduced the notion
of acceptance conditions that should be satisfied at certain
points in the computation. If these conditions are not satisfied,
the program is restored to previous state from where another
recovery block is executed. Checkpointing and recovery based
approaches provide mechanism to restore the program to a
previous checkpoint. Dijkstra [8] introduced an approach for
specifying (i.e., identifying what the program should provide
irrespective of how it is achieved) one-type of fault-tolerance,
namely stabilization [8]. A stabilizing program partitioned
the state space of the program into legitimate states (predicate S
in Figure 1(a)) and other states. It is required that (1) starting
from any state in S, the program always stays in S and (2)
starting from any state in ¬S, the program recovers to a state
in S.

Although stabilization is desirable for many programs, it
is not suitable for some programs. For example, it may be
impossible to provide recovery from all possible states in ¬S.
Also, it may be desirable to satisfy certain safety properties
during recovery. Arora and Gouda [4] introduced another
approach for formalizing a fault-tolerant system that ensures
convergence in the presence of transient faults (e.g., soft errors,
loss of coordination, bad initialization), say f. That is, from
any state/configuration, the system recovers to its invariant S,
in a finite number of steps. Moreover, from its invariant, the
executions of the system satisfy its specifications and remain in
the invariant; i.e., closure. They distinguish two types of fault-
tolerant systems: masking and nonmasking. In the former the
effects of failure is completely invisible to the application. In
other words, the invariant is equal to the fault-span (S = FS
in Figure 1(b)). In the latter, the fault-affected system may
violate the invariant but the continued execution of the system
yields a state where the invariant is satisfied (See Figure 1(b)).

The approach by Arora and Gouda intuitively requires that
after faults stop occurring, the program provides the original
functionality. However, in some cases, restoring the program
to original legitimate states so that it satisfies the subsequent
specification may be impossible. Such a concept has been
considered in [15] where authors introduce the notion of
graceful degradation. In graceful degradation (cf. Figure 1(c)),
a system satisfies its original specification when no faults have
occurred. After occurrence of faults (and when faults stop
occurring), it may not restore itself to the original legitimate
states (S in Figure 1(c)) but rather to a larger set of states (S′
in Figure 1(c)) from where it satisfies a weaker specification.
In other words, in this case, the system may not satisfy the
original specification even after faults have stopped.

In some instances, especially where the perturbations are
security related, it is not sufficient to restore the program to
its original (or somewhat degraded) behavior. The notion of
multi-phase recovery was introduced for such programs [7].
Specifically, in these programs, it is necessary that recovery
is accomplished in a sequence of phases, each ensuring that
the program satisfies certain properties. One of the properties
of interest is strict 2-phase recovery, where the program first
recovers to states Q that are strictly disjoint from legitimate
states. Subsequently, it recovers to legitimate states (See Figure
1(d)).

The goal of auditable restoration is motivated by combining
the principles of the strict 2-phase recovery and the principles

∗ This work is supported by NSF CNS 1329807, NSF CNS 1318678, and
XPS 1533802.
of fault-tolerance. Intuitively, the goal of auditable restoration is to classify system perturbations into faults and auditable events, provide fault-tolerance (similar to that in Figure 1(b)) to the faults and ensure that strict 2-phase recovery is provided for auditable events. Unfortunately, this cannot be achieved for arbitrary auditable events since in [7], it has been shown that adding strict 2-phase recovery to even a centralized program is NP-complete. Our focus is on auditable events that are (immediately) detectable. Faults may or may not be detectable. Unfortunately, adding strict 2-phase recovery to a program even in centralized systems has been shown to be NP-complete.

B. Goals of Auditable Restoration

In our work, we consider the case that the system is perturbed by possible faults and possible tampering that we call as auditable events. Given that both of these are perturbations of the system, we distinguish between them based on how and why they occur. By faults, we mean events that are random in nature. These include process failure, message losses, transient faults, etc. By auditable events, we mean events that are deliberate in nature for which a detection mechanism has been created. Among other things, the need for managing such events arises due to conflicting nature of system requirements. For example, consider a requirement that states that each process in a distributed system is physically secure. This requirement may conflict with another requirement such as each node must be provided emergency access (e.g., for firefighters). As another example, consider the requirement that each system access must be authenticated. This may conflict with the requirement for (potential) unauthorized access in a crisis. Examples of this type are well-known in the domain of power systems, medical record systems, etc., where the problem is solved by techniques such as writing down the password in a physically secure location that can be broken into during crisis or by allowing unlimited access and using logs as a deterrent for unauthorized access. Yet another example includes services such as gmail that (can) require that 'one-time' password for gmail or violation of physical security of a process is detectable. Likewise, the passwords stored in a physically secure location can be different from those used by ordinary users making them detectable.

In this paper, we consider the case where the auditable events are immediately detectable. This is the case in all scenarios discussed above. For example, the use of ‘one-time’ password for gmail or violation of physical security of a process is detectable. Likewise, the passwords stored in a physically secure location can be different from those used by ordinary users making them detectable.

In this paper, for simplicity, we assume that all processes are relevant.

We require that if some process is affected by an auditable event, then eventually all (respectively, relevant) processes in the system are aware of this auditable event. For example, if one process is physically tampered with then it will automatically cause the tampering to be detected at other processes. This would allow the possibility that they only provide the ‘emergency’ services and protect the more sensitive information. This detection must occur even in the presence of faults (except those that permanently fail all processes that were aware of auditable event)\(^2\). We denote such states as auditable states in that all processes are aware of a new auditable event (\(S_2\) in Figure 1(e)).

- After all processes are aware of the auditable event, there exists at least one process that can begin the task of restoring the system to a normal state, thereby clearing the auditable event. This operation may be automated or could involve human-in-the-loop. However, the system should ensure that this operation cannot be initiated until all processes are aware of the auditable event. This ensures that any time the normal operation is restored, all processes are aware of the auditable event.

- If an operation to restore the system to normal state succeeds even if it is perturbed by faults such as failure of processes. However, if an auditable event occurs while the system is being restored to the normal state, the auditable event has a higher priority, i.e., the operation to restore to normal state would be canceled until it is initiated at a later time.

Observe that such a solution is a variation of strict 2-phase recovery. When an auditable event occurs, the system is guaranteed to recover to a state where all processes are aware of this event (\(S_2'\) in Figure 1(e)). And, subsequently, the system recovers to its normal legitimate states (\(S_1\) in Figure 1(e)). Our solution has the following properties:

- Our solution is self-stabilizing. If it is perturbed to an arbitrary state, it will recover to a state from where all future auditable events will be handled correctly.

- Our solution can be implemented with a finite state space. The total state does not increase with the length of the computation. It is well-known that achieving finite state space for self-stabilizing programs is difficult [20].

- Our solution ensures that if auditable events occur at multiple processes simultaneously, it will be treated as one event restoring the system to the auditable state. However, if an auditable event occurs after the system restoration to normal states has begun (or after system restoration is complete), it will be treated as a new auditable event.

- After the occurrence of an auditable event, the system recovers to the auditable state even if it is perturbed by failure of processes, failure of channels, as well as certain transient faults.

\(^2\)If all processes that are (directly or indirectly) aware of the auditable events fail then it is impossible to distinguish this from the scenario where these processes fail before the auditable events.
• No process can initiate the restoration to normal operation unless the system is in an auditable state. In other words, in Figure 1(e), a process cannot begin restoration to normal state unless the system was recently in $S2$.

• After the system restores to an auditable state and some process initiates the restoration to normal operation, it completes correctly even if it is perturbed by faults such as failure of processes or channels. However, if it is perturbed by an auditable event, the system recovers to the auditable state again.

Organization of the paper. The rest of the paper is organized as follows: In Section II, we present the preliminary concepts of stabilization and fault-tolerance and introduce the notion of auditable restoration in Section III. In Section IV, we explain auditable restoration protocol and illustrate it with an example in Section V. Section VII provides related work, and, finally, we conclude in Section VIII.

II. Preliminaries

In this section, we first recall the formal definitions of programs, faults, auditable events, and self stabilization adapted from [4], [8], [12]. We then formally define our proposed auditable restoration mechanism.

Definition 1 (Program): A program $p$ is specified in terms of a set of variables $V$, each of which is associated with a domain of possible values, and a finite set of actions of the form

$$\langle \text{name} \rangle :: \langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle$$

where $\text{guard}$ is a Boolean expression over program variables, $\text{statement}$ updates the program variables.
For such a program, say $p$, we define the notion of state, state space and transitions next.

**Definition 2 ((Program) State and State Space):** A state of the program is obtained by assigning each program variable a value from its domain. The state space (denoted by $S_p$) of such program is the set of all possible states.

**Definition 3 (Enabled):** We say that an action $g \rightarrow st$ is enabled in state $s$ if $g$ evaluates to true in $s$.

**Definition 4 (Transitions corresponding to an action):** The transitions corresponding to an action $ac$ of the form $g \rightarrow st$ (denoted by $\delta_p$) is a subset of $S_p \times S_p$ and is obtained as $\{(s_0, s_1)|g \text{ is true in } s_0 \text{ and } s_1 \text{ is obtained executing } st \text{ in state } s_0\}$.

**Definition 5 (Transitions corresponding to a program):** The transitions of program $p$ (denoted by $\delta_p$) consisting of actions $ac_1, ac_2, \cdots ac_m$ is

$$\delta_p = \bigcup_{i=1}^{m} ac_i \cup \{(s_0, s_0)|ac_1, ac_2 \cdots ac_m \text{ are not enabled in } s_0\}.$$ 

**Remark 1:** Observe that based on the above definition, for any state $s$, there exists at least one transition in $\delta_p$ that originates from $s$. This transition may be of the form $(s, s)$.

For subsequent discussion, let the state space of program $p$ be denoted by $S_p$ and let its transitions be denoted by $\delta_p$.

**Definition 6 (Computation):** We say that a sequence $(s_0, s_1, s_2, \ldots)$ is a computation iff

- $\forall j \geq 0 :: (s_j, s_{j+1}) \in \delta_p$

**Definition 7 (Closure):** A state predicate $S$ is closed in $p$ iff $\forall s_0, s_1 \in S_p :: (s_0 \in S \land (s_0, s_1) \in \delta_p) \Rightarrow (s_1 \in S)$.

**Definition 8 (Invariant):** A state predicate $S$ is an invariant of $p$ iff $S$ is closed in $p$.

**Remark 2:** Normally, the definition of invariant (legitimate states) also includes a requirement that computations of $p$ that start from an invariant state are correct with respect to its specification. The theory of auditable restoration is independent of the behaviors of the program inside legitimate states. Instead, it only focuses on the behavior of $p$ outside its legitimate states. We have defined the invariant in terms of the closure property alone since it is the only relevant property in the definitions/theorems/examples in this paper.

**Definition 9 (Convergence):** Let $S$ and $T$ be state predicates of $p$. We say that $T$ converges to $S$ in $p$ iff

- $S \subseteq T$,
- $S$ is closed in $p$,
- $T$ is closed in $p$, and
- For any computation $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$ of $p$ if $s_0 \in T$ then there exists $l$ such that $s_l \in S$.

**Definition 10 (Stabilization):** We say that program $p$ is stabilizing for invariant $S$ if $S_p$ converges to $S$ in $p$.

**Definition 11 (Faults):** Faults for program $p = \langle S_p, \delta_p \rangle$ is a subset of $S_p \times S_p$; i.e., the faults can perturb the program to any arbitrary state.

**Definition 12 (Auditable Events):** Auditable events for program $p = \langle S_p, \delta_p \rangle$ is a subset of $S_p \times S_p$.

**Remark 3:** Both faults and auditable events are a subset of $S_p \times S_p$, i.e., they both are a set of transitions. However, as discussed earlier, the goal of faults is to model events that are random in nature for which recovery to legitimate states is desired. By contrast, auditable events are deliberate. Additionally, there is a mechanism to detect auditable events and, consequently, we want an auditable restoration technique when auditable events occur.

**Definition 13 (F-Span):** Let $S$ be the invariant of program $p$. We say that a state predicate $T$ is a f-span of $p$ from $S$ iff the following conditions are satisfied: (1) $S \subseteq T$, and (2) $T$ is closed in $f \cup \delta_p$.

### III. Defining Auditable Restoration

In this section, we formally define the notion of auditable restoration. The intuition behind this is as follows: Let $S_1$ denote the legitimate states of the program. Let $T$ be a fault-span corresponding to the set of faults $f$. Auditable events perturb the program outside $T$. If this happens, we want to ensure that the system reaches a state in $S_2$. Subsequently, we want to restore the system to a state in $S_1$.

**Definition 14 (Auditable Restoration):** Let $ae$ and $f$ be auditable events and set of faults, respectively, for program $p$. We say that program $p$ is an auditable restoration program with auditable events $ae$ and faults $f$ for invariant $S_1$ and auditable state predicate $S_2$ iff there exists $T$ such that

- $T$ converges to $S_1$ in $p$,
- $T$ is closed in $\delta_p \cup f$,
- For any sequence $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$ s.t.
  - $s_0 \in T$,
  - $(s_0, s_1) \in ae$,
  - $s_1 \not\in T$,
  - $\forall l :: (s_l, s_{l+1}) \in (\delta_p \cup \delta_f \cup f)$,
  - $\exists w : \forall r : r \geq w \Rightarrow (s_r, s_{r+1}) \in \delta_p$,
  - $\forall x :: (s_x, s_{x+1}) \in ae$,
  - $(\exists m, n :: ((m > n \geq x) \land (s_m \in S_2) \land (s_n \in S_1)))$

### IV. Auditable Restoration for Distributed Programs

In this section we explain our proposed auditable restoration protocol for distributed programs.

As mentioned earlier, the auditable restoration protocol consists of several processes. Each process is potentially capable of detecting an auditable event. However, only a subset of processes is capable of initiating the restoration operation. This captures the intuition that clearing of the auditable event is restricted to authorized processes only and can possibly involve human-in-the-loop. For simplicity, we assume that there is a unique process assigned this responsibility and the failure of this process is handled with approaches such as leader election [17].

The auditable restoration protocol is required to provide the following functionalities: (1) in any arbitrary state where no
auditable event exists, the system reaches a state from where subsequent restoration operation completes correctly, (2) in any arbitrary state where some process detects the auditable event, eventually all processes are aware of this event, (3) some process in the system can detect that all processes have been aware of the auditable event, and (4) the process that knows all processes are aware of the auditable event starts a new restoration operation.

The auditable restoration protocol works as follows: It utilizes a stabilizing silent tree rooted at the leader process, i.e., it reaches a fixpoint state after building the tree even though individual processes are not aware of reaching the fixpoint. Several tree construction algorithms (e.g., [2]) can be utilized for this approach. The auditable restoration protocol is superimposed on top of a protocol for tree reconstruction, i.e., it only reads the variables of the tree protocol but does not modify them. The only variables of interest from the tree protocol are $P_j$ (denoting the parent of $j$ in the tree) and $l_j$ (identifying the id of the process that $j$ believes to be the leader in the tree). Additionally, we assume that whenever a tree action is executed, it notifies the auditable restoration protocol so it can take the corresponding action. Since the tree protocol is silent, this indicates that some tree reconstruction is being done due to faults.

In addition, the program maintains the variable $otsn_j$ and $ctsn_j$. Intuitively, they keep track of the number of auditable events that $j$ has been aware of and the number of auditable events after which the system has been restored by the leader process. Additionally, each process maintains $st_j$ that identifies its state, $sn_j$ that is a sequence number and $res_j$ that is a fixpoint. Several tree construction algorithms (e.g., [2]) can be utilized for this approach. We assume that $otsn_j$ is superimposed on top of a protocol for tree reconstruction, i.e., it only reads the variables of the tree protocol but does not modify them. The only variables of interest from the tree protocol are $P_j$ (denoting the parent of $j$ in the tree) and $l_j$ (identifying the id of the process that $j$ believes to be the leader in the tree). Additionally, we assume that whenever a tree action is executed, it notifies the auditable restoration protocol so it can take the corresponding action. Since the tree protocol is silent, this indicates that some tree reconstruction is being done due to faults.

In addition, the program maintains the variable $otsn_j$ and $ctsn_j$. Intuitively, they keep track of the number of auditable events that $j$ has been aware of and the number of auditable events after which the system has been restored by the leader process. Additionally, each process maintains $st_j$ that identifies its state, $sn_j$ that is a sequence number and $res_j$ that is a fixpoint. Several tree construction algorithms (e.g., [2]) can be utilized for this approach. We assume that $otsn_j$ is superimposed on top of a protocol for tree reconstruction, i.e., it only reads the variables of the tree protocol but does not modify them. The only variables of interest from the tree protocol are $P_j$ (denoting the parent of $j$ in the tree) and $l_j$ (identifying the id of the process that $j$ believes to be the leader in the tree). Additionally, we assume that whenever a tree action is executed, it notifies the auditable restoration protocol so it can take the corresponding action. Since the tree protocol is silent, this indicates that some tree reconstruction is being done due to faults.
this action is a high priority action and executes concurrently with every other action. Hence, for simplicity, we do not show its addition to the rest of the actions. When the leader process is notified of the auditable event, in action AR3, it changes its state to $\bot$ and propagates $\bot$ towards its children. Also, in action AR3, the leader sets its res variable to 1 by using $\min(\text{res}.j + 1, 1)$ function. (Instead of setting res value to 1 directly, we utilize this function because it would simplify the bounding of otsn, ctsn and sn variables in [14].) Actions AR2 and AR3 also need to coordinate with an application that uses this reset protocol to achieve auditable restoration. In particular, these actions should be synchronized with the application so that the application restores itself to an auditable state (i.e., state in S2 in Figure 1(e)). Action AR4 propagates $\bot$ towards the leaves. In action AR5, when a leaf receives $\bot$, it changes its state to $\top$ and propagates it towards the leader. Consider that, action AR5 detects whether all processes have participated in the current $\perp$ wave. This detection is made possible by letting each process maintain the variable res that is true only if its neighbors have propagated that wave. In particular, if process $j$ has completed a $\perp$ wave with res false, then the parent of $j$ completes wave with the res false. It follows that when the leader completes the wave with the res true, all processes have participated in that wave. This action also increments the ctsn value. However, if the leader fails when its state is $\bot$ and some process with state $\top$ becomes the new leader, this assumption will be violated. Action AR3 guarantees that, even if the leader fails, the state of all processes will eventually become $\top$ and the leader process will be aware of that.

Action AR6 broadcasts the changes of the ctsn value so that the other processes can also restore to their legitimate states. We assume that, similar to action AR2, action AR6 is also a high priority action and executes concurrently with the other actions.

When the leader ensures that all processes are aware of the auditable events, a human input can ask the leader to initiate a restoration wave to recover the system to its legitimate state. Therefore, in action AR7, the leader initiates a distributed restoration wave, marks its state as restore, and propagates the restoration wave to its children.

When a process $j$ receives a restoration wave from its parent, in action AR8, $j$ marks its state as restore and propagates the wave to its children. Actions AR7 and AR8 also need to coordinate with the application so that the application state is restored to a stable state (i.e., state in S1 in Figure 1(e)). When a leaf process $j$ receives a restoration wave, $j$ restores its state, marks its state as stable, and responds to its parent. In action AR9, when the leader receives the response from its children, the restoration wave is complete. Action AR9, like action AR5, detects if all processes participated in the current restoration wave by utilizing the variable res. To represent action AR10, first, we define lc.$j$ as follows:

$$\text{lc}.j = ((\text{st}.(P.j) = \text{restore} \land \text{st}.j = \text{restore}) \Rightarrow \text{sn}.j := \text{sn}.(P.j) \land \text{st}.(P.j) = \text{stable} \Rightarrow (\text{st}.j = \text{stable} \land \text{sn}.j = \text{sn}.(P.j)) \land (\text{st}.(P.j) = \perp \land \text{st}.j = \perp) \Rightarrow \text{sn}.j = \text{sn}.(P.j) \land \text{st}.(P.j) = \top \Rightarrow (\text{st}.j = \top \land \text{sn}.j = \text{sn}.(P.j)))$$

Action AR10 guarantees the self-stabilization of the protocol by ensuring that no matter what the initial state is, the program can recover to legitimate states from where future restoration operations work correctly. Finally, if any tree construction algorithm is called to reconfigure the tree and affects process $j$, action AR11 resets the res variable of process $j$.

V. AN ILLUSTRATIVE EXAMPLE

In this section, we present an example to better understand the auditable restoration protocol proposed in Figure 2. In this example, we assume that the system consists of 6 processes $l, i, j, k, m,$ and $n$. Also, we assume that process $l$ is selected as the leader of the system.

Let the initial state be as shown in Figure 3(a). For simplicity, we assume that $sn$, otsn, ctsn, and res values are zero at the beginning. Also, assume that process $j$ has detected an auditable event. Hence, it increments its otsn value by executing action AR1. The computation proceeds as follows:

- The other nodes get notified of the auditable event and increment their otsn value by executing action AR2. The leader $l$ also gets notified of the auditable event by executing action AR2. Therefore, it initiates a $\perp$ wave by executing action AR3 and increments its otsn and sn values (See Figure 3(b)).

- The $\perp$ wave propagates through the system to the leaves. When a leaf, e.g. process $k$, receives it, the leaf first executes action AR4 and changes its state to $\top$. Then, it executes action AR5 and changes its state to $\top$ and propagates the $\top$ wave towards the leader $l$ (See Figure 3(c)).

- When the leader $l$ receives the $\top$ wave, all processes in the system are aware of the auditable event. Moreover, since res.$l$ is 1, none of the processes in the system has failed. At this point, the leader $l$ ensures that the system is in S2. Therefore, it initiates a restoration wave by executing action AR7 (See Figure 3(d)). Also action AR6 executes in concurrent with the other actions since the leader $l$ has updated ctsn.$l$ value by otsn.$l$ value.

- The restoration wave propagates through the system towards the leaves. When the leaf $j$ receives this wave, first, it executes action AR8 and changes its state to restore. At this point, the system has been recovered to $T$ (See Figure 1(e)). Then, since all the neighbors of process $j$ have updated their sn values, process $k$ initiates a stable wave by executing action AR9 (See Figure 3(e)).

- When the leader $l$ receives the stable wave, the state of all processes in the system is stable. This completes the recovery of the system into S1 (See Figure 3(f)).

VI. PROOF OF CORRECTNESS FOR AUDITABLE RESTORATION PROTOCOL

In this section, we, first, explain different types of faults that may happen in the system and then prove the correctness of our proposed protocol.
Fig. 3. A step by step example illustrating the auditable restoration protocol.
A. Fault Types

We assume that the processes are in the presence of (a) fail-stop faults and (b) transient faults. If a process fail-stops, it cannot communicate with the other processes and a tree correction algorithm needs to reconfigure the tree. Transient faults can perturb all variables (e.g., sn, st, etc.) except otsn and ctsn values. For reason of space, we discuss the corruption of the otsn and ctsn values in [14], where we bound these values. Moreover, we assume that all the faults stop occurring after some time. We use $f$ to denote these faults in the rest of this paper.

B. Proof of Correctness

To show the correctness of our protocol, we define the predicates $T$ and $AS$, restoration state predicate $S2$, and invariant $S1$ in the following and use them for subsequent discussions.

$S1: \forall j, k: ((st. j = stable \lor st. j = restore) \land lc. j \land (P. j \text{ forms the tree}) \land (otsn. j = ctsn. k) \land (l. j = \{\text{leader of the parent tree}\})$.

$T: \forall j, k: otsn. j = ctsn. k \land (st. j = restore \lor st. j = stable) \land (P. j = j \land st. j = T \Rightarrow otsn. k \geq otsn. j) \land \exists x: (otsn. j = max(otsn. j) \lor (otsn. j > max(otsn. j)))$.

In $S1$ the state of all processes is either stable or restore and all otsn and ctsn values are equal. If some faults occur but no auditable events, the system goes to $T$ where all otsn and ctsn values are still equal. In this case, the state of the processes cannot be perturbed to $\bot$ or $T$. If auditable events occur, the system goes to $AS$ where the otsn and ctsn values are changed and the max(otsn) is always greater than and equal to max(ctsn). When all the states are changed to $\bot$ or $T$, all processes are aware of the auditable events and the system is in $S2$. Also, the constraint $otsn. k \geq otsn. j$ in the definition of $S2$ means that when the state of the leader is $T$, the rest of the processes are aware of the auditable events that has caused the leader to change its otsn.

Theorem 1: Upon starting at an arbitrary state in $T$, in the absence of faults and auditable events, the system is guaranteed to converge to a state in $S1$.

Proof: Since there is no auditable event in the system, action $AR1$ cannot execute and the otsn and ctsn values do not change. As a result, the system remains in $T$ and executing actions $AR5$, $AR9$, and $AR10$ converges the system to $S1$.

Let $f$ be the faults identified in Section VI-A. Then, we have:

Theorem 2: $T$ is closed in actions ($AR2$–$AR11$) $\cup f$.

Proof: We assume that faults cannot perturb the otsn and ctsn values. Additionally, action $AR1$ does not execute to change the otsn value. This guarantees that actions $AR2$–$AR6$ cannot execute to change the otsn or ctsn values. Hence, when a system is in $T$, in the absence of auditable events, it remains in $T$. Moreover, faults cannot perturb the system to a state outside of $T$, thereby closure of $T$.

Corollary 1: Upon starting at an arbitrary state in $T$, in the presence of faults but in the absence of auditable events, the system is guaranteed to converge to a state in $S1$.

Lemma 1: Starting from a state in $T$ where the otsn value of all processes equal $x$, if at least one auditable event occurs, the system reaches a state where $\forall j: otsn. j \geq x + 1$.

Proof: This lemma implies that, if there is an auditable event in the system and at least one process detects it and increments its otsn value, eventually all processes will be aware of that auditable event.

When the system is in $T$, all the otsn values are equal to $x$. After detecting an auditable event, a process increments its otsn value by executing action $AR1$ to $x + 1$. Consequently, using action $AR2$, every process gets notified of the auditable event and increments its otsn value. If some other process detects more auditable events in the system, it increments its otsn value and notifies the other processes of those auditable events. Hence, the otsn value of all processes is at least $x + 1$.

Theorem 3: Starting from any state in $AS$ – $T$ where $max(otsn. j) > max(ctsn. j)$ and the auditable events stop occurring, the system is guaranteed to reach a state in $S2$.

Proof: When there is a process whose otsn value is greater than all the ctsn values in the system, the state of the system is one of the following:

- there is at least one process that is not aware of all the auditable events occurred in the system, and
- all processes are aware of all auditable events occurred but their states are different.

The first case illustrates that all otsn values are not equal. Consequently, action $AR2$ executes and makes all otsn values equal. When the leader gets notified of the auditable events, it initializes a $\bot$ wave by executing action $AR3$. This wave propagates towards the leaves by executing action $AR4$. When a leaf receives the $\bot$ wave, it changes its state to $T$ by executing action $AR5$ and propagates the $T$ wave towards the leader. Note that, action $AR2$ executes concurrently with the other actions. Thus, all otsn values will eventually become equal. The second case shows that all otsn values are equal but the state of the leader is not changed to $T$ yet. Therefore, when all otsn values are equal and the auditable events stop occurring, the leader executes action $AR5$ and changes its state to $T$, thereby reaching $S2$. Consider that, if a process fails and causes some changes in the configuration of the tree, the res variable of its neighbors would be reset to false and the leader will eventually get notified of this failure by executing action $AR5$. Thus, the leader initializes a new $\bot$ by executing action $AR5$. Moreover, if the leader fails when its state is $\bot$ and another process, say $j$, whose state is $T$ becomes the new leader, the guard of action $AR2$ becomes true since $otsn. j > ctsn. j$. In this case, the new leader initializes a new $\bot$ wave to ensure that when the state of the leader is $T$, the state of all the other processes in the system is also $T$.

Theorem 4: Starting from a state in $T$ and the occurrence of at least one auditable event, the system is guaranteed to reach $S2$ even if auditable events do not stop occurring.
Proof: occurring, at least, one auditable event, some process, say \( j \), detects it and increments its \( otsn \) by executing action \( AR1 \). Following Lemma 1, all processes will eventually get notified of the auditable event. Hence, all \( otsn \) values will be equal and, following Theorem 3, the system will reach a state where the state of all processes is \( \top \) or \( \bot \). Also if the state of the leader is \( \top \), we can ensure that all processes have been notified of the auditable event, thereby reaching a state in \( S2 \). Consider that, even if the auditable events continue occurring, the state of processes does not change and the system remains in \( S2 \).

Theorem 5: Starting from a state in \( T \) and in the presence of faults and auditable events, the system is guaranteed to converge to \( S1 \) provided that faults and auditable events stop occurring.

Proof: Following Theorem 4, if faults and auditable events occur, the system reaches a state in \( S2 \) where all \( otsn \) values are equal and the leader process is aware that all processes have been notified of the auditable events. In this situation, the \( ctsn \) value of the leader is equal to its \( otsn \) value (by executing the statement \( ctsn,j \) := \( otsn,j \) in action \( AR5 \)). Consequently, the leader can start a restoration wave by executing action \( AR7 \). Moreover, the other processes can concurrently execute action \( AR6 \), increase their \( ctsn \) value, and propagate the restoration wave by executing action \( AR8 \). When all \( ctsn \) values are equal, the system is in \( T \) and following Theorems 1 and 2, the system converges to \( S1 \).

Observation 1: The system is guaranteed to recover to \( S2 \) in the presence of faults and auditable events. As long as auditable events continue occurring, the system remains in \( S2 \) showing that all processes are aware of the auditable events. When the auditable events stop occurring, the system converges to \( S1 \), where the system continues its normal execution.

VII. RELATED WORK

Stabilizing Systems. There are numerous approaches for recovering a program to its set of legitimate states. Arora and Gouda’s distributed reset technique introduced in [5] is directly related to our work and ensures that after completing the reset, every process in the system is in its legitimate states. However, their work does not cover faults (e.g., process failure) during the reset process. In [19], we extended the distributed reset technique so that if the reset process is initialized and some faults occur, the reset process works correctly. In [18], Katz and Perry showed a method called global checking and correction to periodically do a snapshot of the system and reset the computation if a global inconsistency is detected. This method applies to several asynchronous protocols to convert them into their stabilizing equivalent, but is rather expensive and insufficient both in time and space.

Moreover, in nonmasking fault-tolerance (e.g., [2], [3]), we have the notion of fault-span too (similar to \( T \) in Definition 14) from where recovery to the invariant is provided. Also, in nonmasking fault-tolerance, if the program goes to \( AS \) \( \rightarrow \) \( T \), it may recover to \( T \). By contrast, in auditable restoration, if the program reaches a state in \( AS \) \( \rightarrow \) \( T \), it is required that it first restores to \( S2 \) and then to \( S1 \). Hence, auditable restoration is stronger than the notion of nonmasking fault-tolerance.

Auditable restoration can be considered as a special case of nonmasking-failsafe multitolerance (e.g., [11]), where a program that is subject to two types of faults \( F_f \) and \( F_n \), provides (i) failsafe fault tolerance when \( F_f \) occurs, (ii) nonmasking tolerance in the presence of \( F_n \), and (iii) no guarantees if both \( F_f \) and \( F_n \) occur in the same computation.

In the context of Cyber-Physical Systems (CPS), in [6], the authors propose a run-time monitoring architecture to maintain safety and progress for a distributed CPS with unreliable communication channels. In this work, all the commands to nodes pass through a central coordinator who keeps track of all the possible states of every node. However, this technique is only applicable to systems where each node has a specific state, called open-loop safe state, and is controlled using a central controller. Also, in [1], the authors propose a recovery mechanism to maintain the stability of the distributed CPS when the communication is lost. Nevertheless, they only limit their technique to communication loss and do not consider any other types of faults nor auditable events.

Tamper Evident Systems. These systems [13], [22], [25] use an architecture to protect the program from external software/hardware attacks. An example of such architecture, AEGIS [25], relies on a single processor chip and can be used to satisfy both integrity and confidentiality properties of an application. AEGIS is designed to protect a program from external software and physical attacks, but did not provide any protection against side-channel or covert-channel attacks. In AEGIS, there is a notion of recovery in the presence of a security intruder where the system recovers to a ‘less useful’ state where it declares that the current operation cannot be completed due to security attacks. However, the notion of fault-tolerance is not considered. In the context of Figure 2, in AEGIS, \( AS \) \( \rightarrow \) \( T \) equals \( \neg S1 \), and \( S2 \) corresponds to the case where tampering has been detected.

Byzantine-Tolerant Systems. The notion of Byzantine faults [21] has been studied in a great deal in the context of fault-tolerant systems. Byzantine faults capture the notion of a malicious user being part of the system. Typically, Byzantine fault is mitigated by having several replicas and assuming that the number of malicious replicas is less than a threshold (typically, less than \( \frac{1}{3} \)rd of the total replicas). Compared with Figure 2, \( T \) captures the states reached by Byzantine replicas. However, no guarantees are provided outside \( T \).

Byzantine-Stabilizing Systems. The notion of Byzantine faults and stabilization have been combined in [10], [23], [24]. In these systems, as long as the number of Byzantine faults is below a threshold, the system provides the desired functionality. In the event the number of Byzantine processes increases beyond the threshold temporarily, the system eventually recovers to legitimate state. Similar to systems that tolerate Byzantine faults, these systems only tolerate a specific malicious behavior performed by the adversary. It does not address active attacks similar to that permitted by Dolev-Yao attacker [9].

In all the aforementioned methods the goal is to restore the system to the legitimate states. In our work, if some auditable events occur, we do not recover the system to the legitimate states. Instead, we recover the system to restoration state where all processes are aware of the auditable events.
occurred. Moreover, in [5], if a process requests a new reset wave while the last reset wave is still in process, the new reset will be ignored. Nevertheless, in our work, we cannot ignore auditable events and all processes will eventually be aware of these events and the system remains in the auditable state as long as the auditable events continue occurring. Also the aforementioned techniques cannot be used to bound $otsm$ value in our protocol.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we presented an algorithm for auditable restoration of distributed systems. This problem is motivated in part by the need for dealing with conflicting requirements. Examples of such requirements include cases where access must be restricted but in some access entirely preventing access is less desirable than some unauthorized access. This also allows one to deal with systems where resistance to tampering/unauthorized access is based on user norms or a legal threat as opposed to a technical guarantee that tampering cannot occur. In other words, such systems provide a In-case-of-emergency-break-glass method for access. By design, these access methods are detectable and called auditable events in our work.

While such an approach often suffices for centralized systems, it is insufficient for distributed systems. In particular, in a distributed system, only one process will be aware of such auditable events. This is unacceptable in a distributed system. Specifically, it is essential that all (relevant) processes detect this auditable event so that they can provide differential service if desired. We denote such states as auditable states.

Auditable events also differ from the typical In-case-of-emergency-break-glass events. Specifically, the latter are one-time events and cannot repeat themselves. By contrast, auditable events provide the potential for multiple occurrences. Also, they provide a mechanism for clearing these events. However, the clearing must be performed after all processes are aware of them and after initiated by an authorized process.

Our program guarantees that after auditable events occur the program is guaranteed to reach an auditable state where all processes are aware of the auditable event and the authorized process is aware of this and can initiate the restoration (a.k.a. clearing) operation. The recovery to auditable state is guaranteed even if it is perturbed by finite number of auditable events or faults. It also guarantees that no process can begin the task of restoration until recovery to auditable states is complete. Moreover, after the authorized process begins the restoration operation, it is guaranteed to complete even if it is perturbed by a finite number of faults. However, it will be aborted if it is perturbed by new auditable events.

Our program is stabilizing in that starting from an arbitrary state, the program is guaranteed to reach a state from where future auditable events will be handled correctly.

We are currently investigating the design and analysis of auditable restoration of System-on-Chip (SoC) systems in the context of the IEEE SystemC language. Our objective here is to design systems that facilitate reasoning about what they do and what they do not do in the presence of auditable events. Second, we plan to study the application of auditable restoration in game theory (and vice versa).

REFERENCES