

Loose Synchronization of Multithreaded Replicas

Claudio Basile, Keith Whisnant, Zbigniew Kalbarczyk, Ravi Iyer
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign, IL 61081
{basilec1, kwhisnan, kalbar, iyer}@crhc.uiuc.edu

Abstract

Although multithreading can improve performance, it is a source of nondeterminism in application behavior. Existing approaches to replicating multithreaded applications either synchronize replicas at interrupt level, at the expense of performance, or use a nonpreemptive deterministic scheduler, at the expense of concurrency. This paper presents a loose synchronization algorithm for ensuring deterministic replica behavior while preserving concurrency. The algorithm synchronizes replica threads only on state updates by enforcing an equivalent order of mutex acquisitions across replicas.

1. Introduction

This paper proposes a *loose synchronization algorithm* (LSA) for handling the nondeterminism induced by multithreading in replica behavior. In contrast with techniques synchronizing replicas at the interrupt level [1], [5], [8], the algorithm synchronizes replica threads on state updates by intercepting mutex requests invoked by threads before accessing shared data in order to enforce an equivalent ordering of mutex acquisitions across replicas. Performance overhead is minimized by preserving concurrency in the execution of application threads—the algorithm interferes with the operating system scheduler only when granting mutexes. This also contrasts with approaches employing nonpreemptive, deterministic schedulers [7], [9], which limit concurrency by allowing only one physical thread to execute at a time.

The idea behind the LSA algorithm can be applied to areas other than active replication, such as log-based rollback recovery, the transaction scheduler of a database server and middlewares providing fault tolerance to CORBA objects.

2. Loose Synchronization Algorithm Overview

In a multithreaded application, shared state updates are serialized via mutex variables. The manner in which threads are granted mutexes is usually nondeterministic and depends on the scheduling algorithm of the operating system. As a result, the programmer cannot make assumptions on the order in which mutexes are acquired. Assuming no *a priori* knowledge of the way replica threads request mutexes, determinism of replica state updates can be achieved by designating a selected replica, the *leader*, to decide the order in which mutexes are granted and to enforce an equivalent order in the other *follower* replicas. All replicas begin executing together, and leader threads freely execute while the order of mutex acquisitions is collected and continuously sent to the followers. The mechanism is such that a follower thread t is

blocked when requesting a mutex m if the order established by the leader for the next acquisition of m either (1) has not yet been received or (2) indicates that m must be acquired first by another thread. Note that the followers enforce the leader's order only with respect to the same mutex. This permits concurrency to be preserved in the execution of follower threads that do not simultaneously acquire the same mutex.

The LSA algorithm uses a leader-follower scheme only to enforce in all replicas the same external behavior, one among all the possible correct behaviors. By agreeing on their external behavior, the replicas can participate in active replication schemes with majority voting.

3. System Model, Definitions, and Assumptions

The system consists of a set of identical multithreaded processes (replicas) running on different nodes and interconnected by means of a network. One replica is designated as the leader; the others are followers. Each replica consists of a set of threads \mathcal{T} and a set of mutexes \mathcal{M} used to protect partitions of shared data (\mathcal{T} and \mathcal{M} can be infinite). Application threads use the functions `lsa_lock` and `lsa_unlock` (replacing the system calls `lock` and `unlock`) to acquire and release a mutex. Two additional functions, `lsa_thr_create` and `lsa_mutex_create`, replace the system calls `thr_create` and `mutex_create`. A FIFO-order reliable multicast and a group membership service are available for leader-to-followers communication. The network does not partition. Application threads are piecewise deterministic (later defined). Replicas share their initial state.

Definition 3.1 (Mutex Acquisition) A triple $(m, t, k) \in \mathcal{M} \times \mathcal{T} \times \mathbb{N}$ denotes a mutex acquisition made by thread t on mutex m through the function `lsa_lock`; this is the k^{th} mutex acquisition made by t .

Expressing mutex acquisitions as triples emphasizes that mutex acquisitions are unique within each replica. To simplify the notation, however, a mutex acquisition (m, t, k) will be referred to as a pair (m, t) . The term k is retrieved by applying a function *index* to the pair (e.g., $k = \text{index}(m, t)$).

Two mutex acquisitions are called *conflicting* if they are made by different threads on the same mutex. In general, the order in which conflicting mutex acquisitions occur can affect the result of the computation.

Definition 3.2 (History) The history H^r of a replica r is the sequence of mutex acquisitions of r 's threads at a given time.

The notation $(m_i, t_i) \stackrel{H^r}{<} (m_j, t_j)$ indicates that (m_i, t_i) temporally precedes (m_j, t_j) in H^r .

Since threads within a replica r execute on the same node, the order of the mutex acquisitions in H^r is determined by the local clock of the node at the time that threads return from `lsa_lock`. Enforcing the leader's history on the followers (under assumption of determinism as defined later) makes the followers behave like the leader. This, however, is a stronger requirement than necessary since only the causal dependencies between mutex acquisitions need to be preserved.

Definition 3.3 (Causal Precedence) *The causal precedence between two mutex acquisitions (m_i, t_i) and (m_j, t_j) in a history H , i.e., $(m_i, t_i) \xrightarrow{H} (m_j, t_j)$, is defined as the transitive closure of the following relation:*

1. $t_i = t_j \wedge (m_i, t_i) \prec^H (m_j, t_j)$, for mutexes acquired by the same thread; or
2. $m_i = m_j \wedge (m_i, t_i) \prec^H (m_j, t_j)$, for conflicting mutex acquisitions.

Causal precedence implies temporal precedence, while the opposite is not necessarily true. The notion of causal precedence between two mutex acquisitions in a multi-threaded process is analogous to the notion of causal precedence between two events in a distributed system [3]. As concurrent events in distributed systems are not causally related, concurrent mutex acquisitions in a multithreaded process are those acquisitions whose actual order of execution does not affect the result of the computation. To preserve concurrency, the LSA allows replicas to schedule concurrent mutex acquisitions independently. Based on the notion of causal precedence, the next definition introduces the causal history of a mutex acquisition.

Definition 3.4 (Causal History) *The causal history of a mutex acquisition (m, t) in a history H is the set $\theta_H(m, t) = \{(m', t') \in H \mid (m', t') \xrightarrow{H} (m, t)\} \cup \{(m, t)\}$.*

The causal history of a given mutex acquisition (m, t) represents all mutex acquisitions upon which (m, t) is causally dependent. Note that a replica history contains all of the replica's mutex acquisitions, while a unique causal history is associated with each mutex acquisition.

The LSA algorithm assumes that threads behave deterministically between two consecutive mutex acquisitions. This is somewhat similar to the piecewise deterministic assumption made by proponents of message-logging checkpointing [6]. While determinism is traditionally expressed in terms of state, the causal history is used as an abstraction to represent a thread's view of the replica's state at the moment of a given mutex acquisition. Moreover, while a thread's behavior in general depends on the inputs it receives, we will not consider these inputs in order to simplify the discussion. Instead, we consider the thread's computation as being purely dependent on the replica's initial state and the interaction, via shared state, with other replica threads. Thread inputs can be incorporated in the model by requiring that corresponding threads of different replicas are supplied the same sequence

of inputs at the same logical time.¹ In this context, we define piecewise thread determinism as follows:

Definition 3.5 (Piecewise Thread Determinism) *A thread t in a replica r is piecewise deterministic iff given its last mutex acquisition (m, t) , the behavior of t is uniquely determined by $\theta_{H^r}(m, t)$ and the replica's initial state S_0^r . In particular, from the initial state (i.e., before its first mutex acquisition), the behavior of t is uniquely determined by S_0^r .*

Because of this definition, outputs emitted by t between a mutex acquisition (m, t) and its next mutex acquisition are a function only of $\theta_{H^r}(m, t)$ and S_0^r . Moreover, race conditions are precluded. The correctness of the LSA algorithm is defined as follows:

Property 3.6 (Correctness) *Given two replicas r_1 and r_2 , two conditions must always hold:*

1. (Safety) *Their causal histories are same: $\forall (m, t) \in H^{r_1} \cap H^{r_2} : \theta_{H^{r_1}}(m, t) = \theta_{H^{r_2}}(m, t)$.*
2. (Liveness) *Any mutex acquisition made by r_1 is eventually² made by r_2 : $(m, t) \in H^{r_1} \implies \Diamond (m, t) \in H^{r_2}$.*

4. Failure Free Behavior

This section assumes that replicas and the reliable multicast layer (i.e., the reliable membership service and the reliable multicast protocol) do not fail. The pseudocode for the LSA algorithm is in Figure 1.³ The functions, variables and definitions used in this pseudocode are given in Table 1.

The leader's history H^l is recorded at the leader l by appending the mutex acquisitions into a fixed-size buffer (`mutex_table`), which, on becoming full, is multicast to followers (via a FIFO-order reliable multicast) and flushed so that new mutex acquisitions can be recorded (`lsa_lock` lines 11–15). The leader's mutex table is also multicasted periodically by `leader_periodic_tx` to guarantee transmission even when there are not enough mutex acquisitions to fill a table.

Followers reconstruct the leader's history by concatenating the mutex tables received from the leader. The leader's history reconstructed by a follower f after receiving n mutex tables $\{mt_1, \dots, mt_n\}$ from the leader l is given by $H^{l,f} = mt_1 \frown mt_2 \frown \dots \frown mt_n$, where \frown is the concatenation operator. In absence of failures, $H^{l,f}$ is a prefix of H^l .

A follower maintains a projection queue⁴ for each mutex m (`proj_q[m]`) that stores the subsequence of $H^{l,f}$ corresponding to mutex acquisitions on mutex m that have yet to be enforced. The follower invokes the function `on_recv_mt` upon receiving a mutex table from the leader to append the

¹In our implementation, this is done by the virtual socket layer [4].

²We use the linear temporal logic symbol \Diamond to denote *eventually*.

³In absence of failures, in `lsa_lock` the lines 21–23 are not executed, and the condition at line 25 is always true; in `on_recv_mt` the condition at line 3 is always false; `on_leader_failed` and `reconf` are not invoked.

⁴Formally, a projection $H|m$ of a history H on a mutex m is the subsequence of the all mutex acquisitions in H conflicting on mutex m such that $(m, t_i) \prec^H (m, t_j)$ iff $(m, t_i) \prec^H (m, t_j)$.

new information to the appropriate projection queue. If a mutex m is not yet in the set of the current replica's mutexes, mutexes, then a new projection queue is created and m is inserted in mutexes (unpack_mt lines 3–6).

```

1: Func lsa_thr_create(t)
2: lock(lsa_mutex)
3: t ← thr_create(f)
4: threads.insert(t)
5: unlock(lsa_mutex)
6: lsa_lock(mc)
7: lsa_unlock(mc)
8: return t
1: Func lsa_mutex_create()
2: lock(lsa_mutex)
3: m ← mutex_create()
4: mutexes.insert(m)
5: if !isLeader then
6:   proj_q[m].create()
7: end if
8: unlock(lsa_mutex)
9: return m
1: Proc lsa_lock(m)
2: lock(lsa_mutex)
3: t ← get_curr_thr()
4: if isLeader then
5:   while owner[m] ≠ ⊥ do
6:     susp_thrs[m].append(t)
7:     suspend(lsa_mutex)
8:     susp_thrs[m].remove(t)
9:   end while
10:  owner[m] ← t
11:  mutex_table.append(m,t)
12:  if #mutex_table = MTS
13:  then
14:    mcast(mutex_table)
15:    mutex_table ← ∅
16:  end if
17: else if can_acquire(m, t) then
18:  owner[m] ← t
19:  proj_q[m].pop()
20: else
21:  if reconfiguring then
22:    if deadlock then
23:      reconft()
24:    end if
25:  end if
26:  if !isLeader then
27:    susp_thrs[m].append(t)
28:    suspend(lsa_mutex)
29:    susp_thrs[m].remove(t)
30:  end if
31:  goto ↓
32: end if
1: Proc lsa_unlock(m)
2: lock(lsa_mutex)
3: owner[m] ← ⊥
4: if isLeader then
5:   resume(susp_thrs[m].head())
6: else if can_sched_next_thr(m)
7:  then
8:   resume(next_thr(m))
9: end if
unlock(lsa_mutex)

```

Figure 1. Pseudocode of the LSA algorithm.

When a follower thread t requests a mutex m by invoking `lsa_lock`, the request is served iff the top entry in `proj_q[m]` is (m, t) and no thread holds m . At that time, (m, t) is extracted from the queue (line 18). Otherwise, t is suspended (line 27). Thread t is resumed when (m, t) reaches the top of `proj_q[m]` and no thread holds m , either when (1) `proj_q[m]` is empty but a new mutex table arrives from the leader and, once unpacked, makes

Table 1. Functions, variables and definitions for the LSA pseudocode.

<code>suspend(m)</code>	Atomically releases a mutex m and suspends the current thread, which holds m when resumed.
<code>resume(t)</code>	Resumes a thread t .
<code>threads</code>	Set of current replica's threads; initially containing only the replica's main thread.
<code>mutexes</code>	Set of current replica's mutexes; initially containing only <code>mc</code> .
<code>isLeader</code>	Boolean variable. True for the leader replica.
<code>lsa_mutex</code>	Global mutex for serializing accesses to LSA code.
<code>mc</code>	Mutex used to enable deadlock detection (see § 5.1).
<code>mutex_table</code>	Queue of size MTS of mutex acquisitions; initially ∅.
<code>proj_q[m]</code>	Array of queues of mutex acquisitions; initially, each queue is ∅.
<code>susp_thrs[m]</code>	Array of lists of suspended threads; initially, each list is ∅.
<code>owner[m]</code>	Array of type $\mathcal{T} \cup \perp$; initially, each entry is \perp .
<code>reconfiguring</code>	Boolean variable. True during reconfiguration mode.
<code>pend_upds</code>	Queue of pending mutex tables; initially ∅.
<code>next_thr(m)</code>	$proj_q[m].head().t$
<code>can_acquire(m, t)</code>	$owner[m] = \perp \wedge proj_q[m] \neq \emptyset \wedge t = next_thr(m)$
<code>can_sched_next_thr(m)</code>	$owner[m] = \perp \wedge proj_q[m] \neq \emptyset \wedge next_thr(m) \in susp_thrs[m]$
<code>deadlock</code>	$\forall m \in mutexes : (proj_q[m] = \emptyset \vee (\exists m' \in mutexes : next_thr(m) \in susp_thrs[m'] \vee next_thr(m) \notin threads))$

`proj_q[m]` have (m, t) as top entry (unpack_mt lines 10–12) or (2) `proj_q[m]` contains an entry (m, t') , with $k = index(m, t')$, immediately preceding (m, t) and thread t' releases m , its k^{th} mutex acquired, through `lsa_unlock` (lsa_unlock lines 6–8). Due to the space limitation, formal proofs of LSA correctness are relegated to [4].

5. Failure Behavior with Error-free Leader-to-Followers Communication

The LSA algorithm introduces asymmetry in replicas (leader and followers) and requires direct communication from leader to followers. This brings about failure modes not present in traditional replication schemes (e.g., [11]). This section analyzes the behavior of the LSA algorithm in the presence of a single, arbitrary failure. The group membership service and the FIFO-order reliable multicast used in the leader-to-followers communication are assumed not to fail (or, equivalently, to mask their Byzantine failures [10]). In this way, nonfaulty followers always have a consistent view of the replicas in the system and always receive the same sequence of messages from the leader.

The architectural setup for the following discussion contains a single, independent voter in the system. The voter is in charge of detecting replica failures—crashes, hangs, and value errors—whether they originate from the application or the LSA code. The voter also excludes faulty replicas from the system (in general, these responsibilities can be placed in other processes outside the voter). The voter does not fail.

Before proceeding, we define two conditions: *deadlock* and *hang*. A deadlock—detected by followers—is the condition in which no mutexes can be acquired (i.e., no thread will return from `lsa_lock`). Deadlock happens when the follower reconstructed leader's history $H^{L,f}$ is not compatible with the replicated application's algorithm. A hang—detected by the voter—is the condition in which an output

is not received from the replica before a timer expires in the voter. We assume that each application thread requests mutexes infinitely often so that a deadlock eventually manifests as a hang to the voter.⁵ Indeed, a deadlock condition at the current time implies that each application thread will be eventually blocked in `lsa_lock`.

5.1. Failure Modes

Leader failures. Errors from the leader can propagate to followers only via the transmission of mutex tables, which is the only leader-to-follower communication channel. The application does not assume any particular mutex acquisition order; thus, a faulty leader cannot cause a correct follower to perform invalid computation, only to diverge or deadlock. If the properties of the reliable multicast service are preserved, all nonfaulty followers receive the same sequence of messages from the leader (even if the leader sends corrupted messages). This guarantees that each pair of nonfaulty followers satisfies the correctness property (as shown in [4]). All nonfaulty followers consequently grant the same causally ordered set of mutexes; thus, if one nonfaulty follower's execution diverges from the leader, then all nonfaulty followers diverge in the same way. Divergent behavior can lead to value errors detected by the voter (if the outputs never differ despite the divergent behavior, then the error has no consequence on the system). In addition to diverging, nonfaulty followers can deadlock. In [4] we prove that if one nonfaulty follower deadlocks, then all nonfaulty followers deadlock. Note that cases such as a leader sending different mutex tables to different followers constitute failures of the reliable multicast layer of the leader and are considered separately in § 6.

If the leader crashes or hangs, it may have sent corrupted mutex tables to the followers before failing, which can lead the followers to either diverge or deadlock as described above. A misbehaving leader can also impersonate a follower, effectively stopping the transmission of mutex tables. Since nonfaulty followers require these messages to make progress, they will eventually deadlock, a condition that the voter detects as a hang.

Follower failures. Corrupted mutex tables from a faulty leader cannot cause a follower to crash—they can result in either divergent behavior or deadlock of the follower. A follower crashing as a result of mishandling faulty data from the leader is treated as a double-failure scenario: a failure in the leader and a failure in the follower caused by an implementation not conforming to the pseudocode in Figure 1. Thus, it can be assumed that a crash detected in a follower is isolated to the failed follower.

While a correct follower does not interact with other replicas, a misbehaving follower can impersonate the leader by sending mutex tables to other replicas. The leader unforgeably signs its messages so that the recipients can always discard messages from unexpected sources.

⁵Long computations can be instrumented with calls to `lsa_lock/unlock` on an artificial mutex to limit the hang manifestation latency.

Deadlock condition. Deadlock defines a situation in which the LSA algorithm in a follower ceases to make progress. This happens when one of the following conditions hold for each projection queue: (1) the projection queue is and will continue to be empty, (2) the thread in the top entry of the projection queue is suspended, or (3) the thread in the top entry of the projection queue does not exist and will never be created. The LSA algorithm checks for deadlock only during reconfiguration (described in § 5.3), when it is known that no new mutex tables will be received.

The first two conditions are easy to check, as is the first clause of the third condition. Checking the second clause of the third condition, however, requires knowledge that the thread in question (thread t) will never be created in the future. Ideally, we would like to drop this part of the condition. However, if the parent of t is executing—but simply has not reached the point at which it creates t —a deadlock could be incorrectly detected. To overcome this problem, the LSA algorithm introduces an artificial mutex `mc` that is acquired through `lsa_lock` each time a new thread is created (see function `lsa_thr_create` in Figure 1). The followers, therefore, contain a projection queue for `mc`, which implicitly identifies the threads that are to create child threads in the future. With `mc` in place, the third condition only needs to check for the existence of the thread. The intuition is that if all projection queues are blocked, then the projection queue corresponding to `mc` is blocked as well and, hence, no thread can be created in the future. A formal description and proof of deadlock conditions are given in [4].

5.2. Failure Detection

The voter takes both a majority vote on replica output values and a majority vote on replica hang conditions on a per-thread basis. Using this information, the voter decides the output to be delivered to the client and identifies any faulty replica and excludes it from the system. If the leader is excluded, the system must be reconfigured. The exclusion of a follower does not require system reconfiguration.

The following categories of replica behavior as observed by the voter can be distinguished: (1) *output*—a replica delivers an output to the voter, (2) *no output*—a replica does not produce an output, and (3) *crash*—detected by the multicast layer, which excludes the offending replica from the system (multicast group) and notifies the remaining replicas and the voter through a view change event.

A voting algorithm is initiated each time the voter receives the first output generated by a replica in response to a client request. At that time, a timer is started to detect replica hangs. Voting occurs either on receiving an output from each replica or on the timer expiration. The possible combinations of leader and follower erroneous behavior (and corresponding voter decisions) induced by a single failure are given in Table 2 (for the faulty leader case) and in Table 3 (for the faulty follower case). In both cases, all nonfaulty replicas always behave in the same manner.

The rule employed by the voter in detecting faulty replicas is the following: (1) if all replicas sent an output, the faulty replica is the one whose output differs from majority output—cases L1 and F1; (2) a replica sending a spurious output is faulty—cases L5 and F4; (3) if there is a single hung replica, that replica is faulty—cases L3 and F2; (4) if a majority of replicas are hung, the leader is faulty—cases L2.⁶ Case L4 (all replicas hung) is indistinguishable from the case in which no output is expected and no replica sends any output. Two solutions are proposed to cope with this case.

Application-specific information embedded in the voter. The voter obtains knowledge as to whether an output is expected to arrive from replicas after a given client request. This knowledge can be derived from the client message contents. For example, for a replicated Apache server, the voter can inspect the HTTP header of the client message and determine whether it is a GET request (a response will follow) or a POST request (no response will follow).

Follower-supported deadlock detection. In this solution, the LSA algorithm supports local deadlock detection during normal operation. When no responses are generated to clients despite open client connections, the voter periodically multicasts a message to followers, forcing them to initiate a self-check for a deadlock condition. The followers communicate the outcome of the check to the voter, which determines the leader as faulty if all followers indicate a deadlock condition. Details of the mechanism used by followers to detect deadlock in response to the voter request are described in [4].

5.3. Reconfiguration

This section considers the reconfiguration of the system after a leader failure. The presented procedure does not require creation of new replicas, since the system is reconfigured around replicas that have not been excluded from the system. The reconfiguration procedure is initiated in each follower upon receiving a view change event from the reliable multicast layer corresponding to the leader leaving the multicast group (function `on_leader_failed` in Figure 1), because the leader either crashed or was terminated by the voter after being detected as faulty. A new leader can be selected after all surviving replicas reach the deadlock condition (defined in § 5.1). The reconfiguration procedure consists of the following steps:

1. The follower continues to execute until it detects a deadlock (`lsa_lock` line 21, `on_leader_failed` line 4).
2. All projection queues are cleared to prepare the replica for resuming the execution (`reconf` lines 2–4). After reaching deadlock, remaining entries in the projection queues indicate a sequence of mutex acquisitions incompatible with the application’s algorithm and must be removed (note that entries already consumed are valid).
3. The follower chooses the new leader from the group membership list, which is assumed to be identical for

⁶In case L2 no output can be delivered to the client; however, after reconfiguration, surviving replicas restart execution (exiting from deadlock) and generate the expected output.

Table 2. Replica behavior under faulty leader.

Case	Expected behavior	Faulty leader’s behavior	Followers’ behavior	Diagnosis
L1	Output	Output	Output	Compute majority value. If the leader is in minority, it is faulty.
L2	Output	Output	No output	Followers are in deadlock. Majority is hung thus the leader is faulty.
L3	Output	No output	Output	The leader is the only hung replica thus it is faulty.
L4	Output	No output	No output	Followers are in deadlock. All replicas are hung thus the leader is faulty.
L5	No output	Output	No output	The leader sent a spurious output thus it is faulty.
L6	No output	Output/No output	Output	Not possible. In nonfaulty replicas (even if contaminated), any mutex acquisition order results in the correct behavior.
L7	No output	No output	No output	No fault has manifested.

Table 3. Replica behavior under faulty follower.

Case	Expected behavior	Faulty follower’s behavior	Correct replicas’ behavior	Diagnosis
F1	Output	Output	Output	Compute majority value. If a follower is in minority, it is faulty.
F2	Output	No output	Output	A follower is the only hung replica thus it is faulty.
F3	Output	Output/No output	No output	Not possible since it violates the single failure assumption.
F4	No output	Output	No output	A follower sent a spurious output thus it is faulty.
F5	No output	Output/No output	Output	Not possible since it violates the single failure assumption.
F6	No output	No output	No output	No fault has manifested.

all replicas so that a deterministic selection rule can be applied (e.g., pick the first). If the follower is not chosen to be the new leader, it waits in deadlock until it receives mutex tables from the new leader. The new leader awakens all of its application threads (`reconf` lines 7–11), so they can execute `lsa_lock` as the leader replica.

If the leader-elect replica executes the reconfiguration procedure faster than the other replicas, these replicas may receive mutex tables from the new leader before they have reached a deadlock, i.e., before step (2). It is necessary, therefore, that the followers buffer the mutex tables received during the reconfiguration mode (`on_recv_mt` lines 3–5). The buffered mutex tables are unpacked into the projection queues after choosing the new leader (`reconf` lines 13–16).

The reconfiguration algorithm presented above ensures correctness between the new leader and any follower [4].

6. Failure Behavior with Byzantine Errors in Leader-to-Followers Communication

This section analyzes the impact of failures in the leader-to-followers multicast communication under the single failure scenario. We continue to assume that the group membership protocol does not fail. Violating the properties of the FIFO-order reliable multicast because of a faulty leader can result in: (1) not sending a mutex table at all, (2) sending a mutex table only to some followers, or (3) sending a mutex table with different contents (or in different orders) to different followers. These cases can cause followers to be inconsistent with each other. Our solution does not require a multicast protocol tolerating Byzantine failures and takes action

only after the voter detects inconsistencies, without incurring extra overhead during normal operation.

Failure Detection. The voter detects replica failures and decides upon system reconfiguration as described below.

1. Detecting a follower crash or a follower spurious output indicates that the follower is the single faulty replica in the system. The system can continue without reconfiguration after the faulty follower is excluded.
2. Detecting a follower hang or a follower value error indicates failure either of the follower or of the leader (which has contaminated the follower). Both the follower and the leader must be excluded from the system, since the two cases are indistinguishable.
3. Detecting only a leader failure indicates that the leader is faulty and must be excluded from the system.
4. Detecting misbehavior of multiple replicas (e.g., crash, hang, value error) indicates that an error in the leader has contaminated the followers. Consequently, the leader is the single faulty replica and must be excluded.⁷

Reconfiguration. We select a subset of the remaining correct followers (i.e., followers whose state is consistent with each other) from which the system can restart.⁸ The reconfiguration procedure starts with all followers sending their state to the voter to determine the largest group of followers whose states agree; those in the largest group will survive the failure, and all other followers will be excluded from the system. For this state comparison to be meaningful, followers need to capture their state when their corresponding threads are at the same point. Because there are no mutex table transmissions during reconfiguration, the followers eventually deadlock, and replica state can be captured after reaching this state. Note that a failure can degrade a system with at least three replicas at most to a single running replica.

7. Related Work

Early works on software-based replication synchronize replicas at the interrupt level. In the *TARGON/32* system, asynchronous events (e.g., UNIX signals) are transformed into synchronous messages delivered to the destination process and its backup [1]. In the *Hypervisor* system a virtual machine layer, beneath the operating system, uses the hardware instruction counter to count the instructions executed between two hardware interrupts [5]. *Delta-4* provides semi-active replication with a leader/follower model plus a preemption synchronization mechanism. When an interrupt arrives at the leader, the leader determines the next preemption point at which it will be served. This information is sent to followers. Replicas are assumed to be fail silent [8].

Some of the issues related to handling nondeterminism due to multithreading have been studied in the context of log-based rollback recovery. In [2] it is suggested adding support

⁷Because of the single-failure assumption, only the leader can crash.

⁸Since (possibly contaminated) correct followers cannot perform an invalid computation, it would be sufficient to choose any follower as the single surviving replica and exclude all the other replicas.

to the Mach operating system to track and to log the order in which threads access locks and semaphores.

Existing solutions to replicate multithreaded applications are based on a nonpreemptive deterministic scheduler. *Eternal's* nonpreemptive deterministic scheduler allows the execution of only one logical thread at a time. As a result, concurrency is significantly limited [7]. *Transactional Drago's* nonpreemptive deterministic scheduler targets transactional applications and allows several transactions to execute concurrently. However, scheduling of another thread can be done only when the running thread reaches a *scheduling point* [9]. Both *Eternal* and *Transactional Drago* can schedule only one physical thread at a time, even if multiple CPUs are available.

8. Conclusions

This paper has proposed a *loose synchronization* algorithm for active replication of multithreaded applications. The algorithm enforces "equivalent" ordering of state changes across replicas and guarantees replica consistency with low overhead. The leader establishes the order of mutex acquisitions and sends it to the followers over the network.

To evaluate the proposed algorithm, a transparent active replication framework has been developed in [4] and used to triplicate the multithreaded Apache server. The measured throughput reduction for retrieving a CGI (Common Gateway Interface)-generated 1KB HTML page was about 25%. For details of this evaluation the reader is referred to [4].

References

- [1] A. Borg et al. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, 1989.
- [2] A. Goldberg et al. Transparent recovery of Mach applications. In *Usenix Mach Workshop*, pages 169–183, 1990.
- [3] O. Babaoglu and K. Marzullo. *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.
- [4] C. Basile, Z. Kalbarczyk, K. Whisnant, and R. Iyer. Active replication of multithreaded applications. Technical Report CRHC-02-01, University of Illinois at Urbana-Champaign, 2002. <http://www.uiuc.edu/~cbasile/lisa>.
- [5] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. on Computer Systems*, 14(1):80–107, 1996.
- [6] E. Elnozahy, D. Johnson, and Y. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical report, Carnegie Mellon University, 1996.
- [7] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the *Eternal* system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.
- [8] P. A. Barrett et al. The *Delta-4* extra performance architecture (XPA). In *FTCS-20*, pages 481–488, 1990.
- [9] S. A. R. Jimenez-Peris, M. Patino-Martinez. Deterministic scheduling for transactional multithreaded replicas. In *SRDS-19*, 2000.
- [10] M. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proc. of the 2nd ACM Conf. on Computer and Comm. Security*, pages 68–80, 1994.
- [11] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.