ACTIVE REPLICATION OF MULTITHREADED APPLICATIONS

Claudio Basile, Zbigniew Kalbarczyk, Ravi Iyer,
Keith Whisnant

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE April 2002	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
Active Replication of Multithreaded Applications

5. FUNDING NUMBERS

6. AUTHOR(S)
Claudio Basile, Zbigniew Kalbarczyk, Ravi Iyer, and Keith Whisnant

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)
Coordinated Science Laboratory
1308 W. Main St.
Urhana, IL 61801

8. PERFORMING ORGANIZATION REPORT NUMBER
UILU-ENG-02-2201
CRHC-02-01

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

limited.

12 b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

14. SUBJECT TERMS active replication, multithreading, determinism, voting, software implemented fault tolerance

15. NUMBER IF PAGES
24

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Active Replication of Multithreaded Applications

Claudio Basile, Zbigniew Kalbarczyk, Keith Whisnant, Ravi Iyer
Center of Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801
{basilecl, kalbar, kwhisnan, iyer}@crhc.uiuc.edu
Technical Report CRHC-02-01

Abstract

Software-based active replication is a well-known technique for providing fault tolerance using space redundancy and fault-masking. However, much of the recent research in software replication has yet to be demonstrated using large, real-world applications, and in particular, multithreaded applications. While multithreading can improve performance, thread scheduling is a source of nondeterminism in application behavior. Existing approaches to replicating multithreaded applications employ either synchronization at interrupt level, at the expense of performance, or a nonpreemptive deterministic scheduler, at the expense of concurrency.

This paper presents a loose synchronization algorithm *for ensuring deterministic behavior of replicas while preserving concurrency. The algorithm synchronizes replica threads only on state updates by intercepting mutex requests. The algorithm is formally specified and the proposed formalism is used to prove correctness of the algorithm in failure-free behavior as well as in presence of errors. To evaluate the proposed algorithm, a transparent active replication framework has been developed and used to replicate the multithreaded version of the Apache web server, a substantial real-world application. Performance for a triplicated, multithreaded Apache is about 23% less than the TCP-based, noninstrumented version of the same multithreaded Apache server.*

Keywords: Active Replication, Multithreading, Nondeterminism, Voting, Software Implemented Fault Tolerance

1. Introduction

In a fault-tolerant replicated system, multiple instances of an application execute on independent hardware so that the system can continue to provide correct service in case of a replica failure. Earlier approaches used dedicated, often proprietary, hardware to achieve efficiency and performance (e.g., [16]). Replication in software aims to be more flexible and less costly by making use of commercial, off-the-shelf (COTS) hardware, but its applicability to real-world systems is somewhat limited. In particular, the extension of replication to multithreaded applications requires further investigation.

This paper proposes a *loose synchronization algorithm* (LSA) for ensuring deterministic behavior of replicas while preserving concurrency. In contrast with current techniques that synchronize replicas at the interrupt level [1], [8], [7], [23], the algorithm synchronizes replica threads on state updates (enforcing an "equivalent" order) by intercepting mutex requests invoked by threads before accessing shared data. Performance overhead is minimized by preserving concurrency in the execution of application threads—the algorithm does not interfere with the operating system scheduler, except when granting mutexes. This is also in contrast with approaches employing nonpreemptive, deterministic schedulers [21], [26], which limit concurrency by allowing only one physical thread to execute at a time.

Although intercepting mutex requests to record the order of state updates has been proposed in the context of rollback recovery [2], it has not been applied to active replication, nor has it been demonstrated on a substantial application. To evaluate the proposed algorithm, a transparent active replication framework has been developed. The framework consists of an implementation of the loose synchronization algorithm, a *virtual socket layer* that provides transparent replication and an adaptive
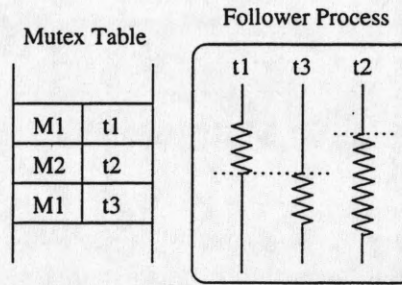
Figure 1. Execution of follower threads.

voter/fanout component that detects errors (crash, hangs, and value errors), excludes faulty replicas, and reliably broadcasts client requests to the replicas. The framework has been used to replicate the multithreaded version of the Apache web server, a substantial real-world application. Results show about a 23% performance degradation for a triplicated Apace web server compared to a noninstrumented, TCP-based version of the same server.

2. Loose Synchronization Algorithm

The proposed *loose synchronization algorithm* (LSA) exploits the fact that nondeterminism in replica behavior is acceptable as long as it does not impact the output produced by replicas and voted upon by the voter. Replica output is typically a function of the replica's state and inputs; hence, suppling the same sequence of inputs and enforcing the same sequence of state updates in each replica guarantees that all replicas produce the same sequence of outputs.

In a multithreaded application, updates to a shared state are serialized through mutex variables (mutual exclusion). The manner in which threads are granted mutexes is usually nondeterministic and depends on the scheduling algorithm used by the operating system. As a result, the programmer cannot usually make assumptions on the order in which mutexes are acquired.

Assuming no *a priori* knowledge of the way mutexes are requested by the replica threads, determinism of replica state updates can be achieved by designating a selected replica, the *leader*, to make decisions on the order in which mutex variables are granted and to enforce an "equivalent" order in other, *follower* replicas. All replicas begin executing together, and leader threads freely execute while the order of mutex acquisitions is collected. The leader's order is continuously sent to the followers, which enforce the same order on their threads. The mechanism is such that a follower thread t can be blocked when acquiring a mutex m if (1) the order established by the leader for the next acquisition of the mutex m has not yet been received or (2) the mutex m needs to be acquired first by another thread (according to the order established by the leader). The follower replicas need to enforce in their threads the order dictated by the leader 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. Figure 1 shows an example of an order of mutex acquisitions sent by the leader to the followers via a mutex table. The follower threads t_1 and t_2 can be executed concurrently (since they acquire different mutexes), while t_1 and t_3 must be serialized.[1]

2.1. 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 process is designated as the leader replica; the others are follower replicas. Each process 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 function lsa_lock (replacing the system call lock) to acquire a mutex. Threads release a mutex using the system call unlock. Two additional functions, create_new_thread and create_new_mutex, are provided to replace, respectively, the system calls thread_create and mutex_create (see Figure 2). The LSA algorithm requires leader and

---

[1] It is assumed that two different mutexes do not protect overlapping (or coincident) shared-memory regions, which is also good programming practice.

followers to exchange information about the mutex acquisition order. A FIFO-order reliable multicast and a reliable group membership service are available. It is also assumed that the network does not partition.

Definition 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 kth mutex acquisition performed by t.*

Expressing mutex acquisitions as triples emphasizes the fact 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 can still be retrieved by applying a function *index* to the pair (e.g., $k = index(m, t)$).

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 2 (History) *A history H^r of replica r is the sequence of mutex acquisitions of r's threads. The notation $(m_i, t_i) \overset{H^r}{<} (m_j, t_j)$ depicts 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 mutex acquisition 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 (Causal Precedence) *Given a history H and two mutex acquisitions (m_i, t_i) and (m_j, t_j) in H, (m_i, t_i) causally precedes (m_j, t_j) in H (i.e., $(m_i, t_i) \overset{H}{\leadsto} (m_j, t_j)$), if and only if one of the following conditions holds:*

1. $t_i = t_j \wedge (m_i, t_i) \overset{H}{<} (m_j, t_j)$; *(mutexes acquired by the same thread);*

2. $m_i = m_j \wedge (m_i, t_i) \overset{H}{<} (m_j, t_j)$; *(conflicting mutex acquisitions);*

3. $\exists (m, t) \in H : (m_i, t_i) \overset{H}{\leadsto} (m, t) \wedge (m, t) \overset{H}{\leadsto} (m_j, t_j)$; *(transitivity of causal dependency).*

Note that causal precedence implies temporal precedence, while the opposite is not necessarily true. The notion of causal precedence between two mutex acquisitions in a multithreaded process is analogous to the notion of causal precedence between two events in a distributed system [4]. As there are concurrent events in distributed systems (i.e., events that are not causally related), there are concurrent mutex acquisitions in a multithreaded process (i.e., acquisitions whose actual order of execution does not affect the result of the computation). The LSA algorithm allows replicas to schedule concurrent mutex acquisitions independently in order to preserve concurrency.

Based on the notion of causal dependency, the next definition introduces the causal history of a mutex acquisition.

Definition 4 (Causal History) *Given a history H and a mutex acquisition (m, t) in H, the* causal history *of (m, t) is the set* $\theta_H(m, t) = \{(m', t') \in H | (m', t') \overset{H}{\leadsto} (m, t)\} \cup \{(m, t)\}$.

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 [11]. 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.

Definition 5 (Piecewise Thread Determinism) *A thread t in a replica r is* piecewise deterministic *if and only if given the 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. From the initial state (i.e., before the first mutex acquisition), the behavior of t is uniquely determined by S_0^r.*

From the above definition, it follows that the behavior and, hence, the outputs emitted by a thread t in a replica r between a mutex acquisition (m, t) and the next mutex acquisition are a function only of $\theta_{H^r}(m, t)$ and S_0^r. Note that the definition precludes race conditions in the replica's code. Only replicated applications whose threads are piecewise deterministic and share the same initial state are considered in this paper. In the context of such a system, the correctness of the LSA algorithm is defined as follows:

Property 1 (Correctness) *Given two replicas r_1 and r_2, two conditions hold:*

1. *(Safety) The causal histories of the two replicas must be the same:* $\forall (m, t) \in H^{r_1}, H^{r_2} : \theta_H^{r_1}(m, t) = \theta_{H^{r_2}}(m, t)$.

2. *(Liveness) Any mutex acquisition within one replica is eventually[2] performed by the other replica:* $(m, t) \in H^{r_1} \implies \Diamond(m, t) \in H^{r_2}$.

2.2. Failure Free Behavior

In the following discussion, we assume 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 shown in Figure 2.[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 by appending the mutex acquisitions into a fixed-size buffer (`mutex_table`). When the leader's mutex table becomes full, the leader multicasts the table to followers (with a FIFO-order reliable multicast), and flushes it so that new mutex acquisitions can be recorded. The leader's mutex table is also multicast periodically by `leader_periodic_tx` in order to guarantee transmission even when there are not enough mutex acquisitions to fill a table.

Conceptually, followers reconstruct the leader's history by concatenating the mutex table updates received from the leader. The leader's history reconstructed by a follower f after receiving n mutex table updates $\{mt_1, \ldots, mt_n\}$ from the leader l is given by $H^{l,f} = mt_1 \frown mt_2 \frown \ldots \frown mt_n$, where $\frown$ is the concatenation operator. Thus, in absence of failures, $H^{l,f}$ is a prefix of H^l.

A follower maintains a projection queue for each mutex m (`proj_queue[m]`) that stores the subsequence of $H^{l,f}$ corresponding to mutex acquisitions on mutex m that have yet to be enforced.[4] The follower invokes the function `on_mt_update`, upon receiving a mutex table update from the leader, to append the new updates 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`.

When a follower thread t requests mutex m by invoking `lsa_lock`, the request is served only if the top entry in `proj_queue[m]` is (m, t). Otherwise, t is suspended—`proj_queue[m]` is empty or its top entry indicates a different thread. Thread t is resumed when (m, t) reaches the top of `proj_queue[m]`: (1) `proj_queue[m]` is empty but a new mutex table update arrives from the leader and, once unpacked, makes `proj_queue[m]` have (m, t) as top entry (`perform_update` lines 9–13) or (2) `proj_queue[m]` contains an entry (m, t'), with $k = index(m, t')$, immediately preceeding (m, t) and thread t' acquires m, as its kth mutex acquisition, through `lsa_lock` (`lsa_lock` lines 14–17).

Proofs for leader–follower correctness and follower–follower correctness are given in the Appendix.

---

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

[3]In absence of failures, in `lsa_lock` the lines 20–24 are not executed, and the condition at line 25 is always true. Moreover, in `on_mt_update` the condition at line 3 is always false, and `on_leader_failed` and `reconfigure` are not invoked.

[4]Formally, 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 $(m, t_i) \overset{H|m}{<} (m, t_j)$ iff $(m, t_i) \overset{H}{<} (m, t_j)$.

```
1: Function create_new_thread(f)              1: Procedure lsa_lock(m)                        1: Procedure perform_update(update)
2:    lock(lsa_mutex)                         2:    lock(m)                                   2:    for all (m, t) ∈ update do
3:    t ← thread_create(f)                    3:    lock(lsa_mutex)                           3:       if m ∉ mutexes then
4:    threads.insert(t)                       4:    t ← get_curr_thread()                     4:          proj_queue[m].create()
5:    unlock(lsa_mutex)                       5:    repeat                                    5:          mutexes.insert(m)
6:    lsa_lock(mc)                            6:       if isLeader then                       6:       end if
7:    unlock(mc)                              7:          mutex_table.append(m,t)             7:       proj_queue[m].append(m,t)
8:    return t                                8:          if #mutex_table = maxentries then   8:    end for
                                              9:             cast(mutex_table)                9:    for all m ∈ mutexes do
1: Function create_new_mutex()                10:            mutex_table ← ∅                 10:       if can_schedule_next_thread(m) then
2:    lock(lsa_mutex)                         11:         end if                             11:          resume(next_thread(m))
3:    m ← mutex_create()                      12:         exit ← true                        12:       end if
4:    mutexes.insert(m)                       13:      else if can_acquire_mutex(m) then     13:    end for
5:    if ¬isLeader then                       14:         proj_queue[m].pop()
6:       proj_queue[m].create()               15:         if can_schedule_next_thread(m) then 1: Procedure on_leader_failed()
7:    end if                                  16:            resume_thread(next_thread(m))    2:    lock(lsa_mutex)
8:    unlock(lsa_mutex)                        17:         end if                             3:    reconfigurating ← true
9:    return m                                18:         exit ← true                        4:    {If already in deadlock initiate reconfiguration.}
                                              19:      else                                  5:    if deadlock then
1: Procedure suspend_thread(m,t)              20:         if recovering then                 6:       reconfigure()
2:    suspended_threads.insert(t)             21:            if deadlock then                 7:    end if
3:    unlock(lsa_mutex)                       22:               recovery()                    8:    unlock(lsa_mutex)
4:    suspend(m)                              23:            end if
5:    lock(lsa_mutex)                         24:         end if                              1: Procedure reconfigure()
                                              25:         if ¬isLeader then                  2:    for all m ∈ mutexes do
1: Procedure resume_thread(t)                 26:            suspend_thread(m,t)              3:       proj_queue[m] ← ∅
2:    suspended_threads.delete(t)             27:            exit ← false                     4:    end for
3:    resume(t)                               28:         end if                              5:    isLeader ← choose_new_leader()
                                              29:      end if                                6:    if isLeader then
1: Procedure leader_periodic_tx()             30:   until exit                               7:       for all t' ∈ suspended_threads do
2:    while true do                           31:   unlock(lsa_mutex)                         8:          resume_thread(t')
3:       sleep(LEADER_TX_PERIOD)                                                              9:       end for
4:       lock(lsa_mutex)                                                                      10:   else
5:       if mutex_table ≠ ∅ then              1: Procedure on_mt_update(update)               11:      for all u ∈ pending_updates do
6:          cast(mutex_table)                 2:    lock(lsa_mutex)                           12:         on_mt_update1(u)
7:          mutex_table ← ∅                   3:    if reconfigurating then                  13:      end for
8:       end if                               4:       pending_updates.append(update)        14:      pending_updates ← ∅
9:       unlock(lsa_mutex)                     5:    else                                     15:   end if
10:   end while                               6:       perform_update(update)                16:   reconfigurating ← false
                                              7:    end if
                                              8:    unlock(lsa_mutex)
```

Figure 2. Pseudocode of the loose synchronization algorithm.

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

lock(m)	System call to lock a mutex m.
unlock(m)	System call to unlock a mutex m.
suspend(m)	System call to release a mutex m and suspend the current thread. The thread holds m when resumed.
resume(t)	System call to resume a thread t.
sleep(ts)	System call to suspend the current thread for ts seconds.
thread_create(f)	System call to create a new thread, which will execute function f.
get_curr_thread()	Returns the descriptor of the current thread.
cast(msg)	Reliable multicast of msg to followers.
create_new_thread(f)	Invoked by replica's code to create a new thread, which will execute function f.
create_new_mutex()	Invoked by replica's code to create a new mutex.
on_mt_update(m)	Invoked by a follower on receiving a message m from the leader.
leader_periodic_tx()	Periodic transmission of leader's mutex_table to followers.
lsa_lock(m)	Invoked by replica's code to lock a mutex m.
choose_new_leader()	Deterministic rule to choose the new leader from the members of the current view.
on_leader_failed()	Invoked by a follower when the leader leaves the multicast group.
threads	Set of current replica's threads; initially containing the replica's main thread.
mutexes	Set of current replica's mutexes; initially containing mc.
isLeader	Boolean variable.
lsa_mutex	Global mutex used to serialize accesses to LSA code.
mc	Mutex used to serialize accesses to create_new_thread.
mutex_table	Queue of length maxentries of mutex acquisitions; initially empty.
proj_queue[m]	Array of queues of mutex acquisitions; initially empty.
suspended_threads	List of suspended threads; initially empty.
reconfigurating	Boolean variable.
pending_updates	Queue of mutex table updates; initially empty.
$next\_thread(m)$	$proj\_queue[m].head().t$
$can\_acquire\_mutex(m, t)$	$proj\_queue[m] \neq \emptyset \wedge t = next\_thread(m)$
$can\_schedule\_next\_thread(m)$	$proj\_queue[m] \neq \emptyset \wedge next\_thread(m) \in suspended\_threads$
$deadlock$	$\forall m \in mutexes : (proj\_queue[m] = \emptyset \vee proj\_queue[m].head().t \in suspended\_threads$ $\vee\ proj\_queue[m].head().t \notin threads)$

2.3. 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 in failure modes not present in traditional replication schemes (e.g., [28]). In this section, we analyze the behavior of the LSA algorithm in the presence of a single, potentially malicious failure. The group membership service and the FIFO-order reliable multicast employed in the leader-to-followers communication are assumed not to fail. 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. An equivalent assumption is that these protocols can mask their Byzantine failures (e.g., [17] and [27]).

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).

Before proceeding, we define two conditions: *deadlock* and *hang*. A deadlock—detected by followers—is the condition in which no more mutexes can be acquired, i.e., no thread will ever return from `lsa_lock`. Deadlock happens when the reconstructed leader's sequence of mutex acquisitions $H^{l,f}$ 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 also assume that mutexes are requested by replica's code infinitely often so that a replica deadlock eventually manifests as a hang to the voter.[5]

2.3.1. Failure Modes
The failure modes induced by the LSA algorithm are discussed below and are summarized in Table 2.

1. *Leader failures.* Errors from the leader can propagate to followers only via the transmission of mutex table updates (which is the only communication from the leader to followers). Assuming that 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 Theorem 6 in the Appendix). 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. Corollary 1 (in Appendix) guarantees that if one nonfaulty follower deadlocks, then all nonfaulty followers deadlock. Note that cases such as a leader sending different mutex table updates to different followers constitute failures of the reliable multicast layer of the leader and are considered separately in Section 2.4.

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

2. *Follower failures.* Corrupted mutex table updates 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 mishanding 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 a poor implementation that does not conform to the pseudocode in Figure 2). Thus, it can be assumed that a crash detected in a follower is isolated to the failed follower.

---
[5]Long computation periods can be instrumented with calls to `lsa_lock/unlock` on an artificial mutex to limit the hang manifestation latency.

Table 2. Failure modes of the LSA algorithm.

Failure	What can be inferred from:	
	Follower Failure	Leader Failure
Crash.	The follower is faulty.	The leader can have contaminated the system.
Hang.	The leader could be the cause.	The leader can have contaminated the system.
Follower diverges from leader.	The leader could be the cause.	N.A.
Leader sends corrupted (or omits sending) mutex table updates.	N.A.	The leader can contaminate the system.
Impersonation.	The follower can contaminate the system.	The leader causes all replicas to hang.

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

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 (as shown in Theorem 7 in the Appendix): (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, when it is known that no new mutex table updates will be received in the future. [6]

The first two conditions are easy to check, as is the first clause of the third condition. To check 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 thread 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 create_new_thread in Figure 2). 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. This is formally shown in the Appendix.

2.3.2. Failure Detection.

To detect failures, the voter takes both a majority vote on output values produced by replicas and a majority vote on replica hang conditions. 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 (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.

The 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 the reception of an output from each replica or on the timer expiration. The possible combinations of leader and follower failure behavior (and corresponding voter decisions) are given in Table 3 (for the faulty leader case) and in Table 4 (for the faulty follower case). In both cases, all nonfaulty replicas always behave in the same manner.

The rules employed by the voter in detecting faulty replicas can be summarized as follows: (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

---

[6]If the reliable multicast protocol guarantees that a message is delivered at the same view as it is sent, then no mutex table updates will be received during reconfiguration [6]. Some group communication systems only guarantee that a message is delivered at the same view at every process that delivers it [12]. In this case, leader's messages received after the leader leaves the multicast group can be safely discarded.

7

Table 3. Replica behavior under faulty leader.

Case	Expected behavior	Faulty leader behavior	Followers' behavior	Diagnosis
L1	Output	Output	Output	Compute majority value. If leader is in minority then the leader 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, the leader 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, the leader is faulty.
L6	No output	Output/No output	Output	Not possible. The application does not assume any particular mutex acquisition order. Thus, 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 4. Replica behavior under faulty follower.

Case	Expected behavior	Faulty follower behavior	Correct replicas' behavior	Diagnosis
F1	Output	Output	Output	Compute majority value. If the follower is in minority then it is faulty.
F2	Output	No output	Output	The follower is the only hung replica; thus, the follower 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, the follower 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.

replicas are hung, the leader is faulty—cases L2.[7] Case L4 (all replicas hanging) 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 supposed 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). For a replicated CORBA object, the GIOP header of a request message contains a field `response_expected` that is true if and only if a reply message will follow. In general, if necessary, the client can be instrumented to extend the message format to indicate whether a response is going to follow the request.

Follower-supported deadlock detection. In this solution, the LSA algorithm supports local deadlock detection. During periods in which no responses are generated to the 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. The mechanism for followers to detect deadlock in response to the voter message is described in Section 8.3 (in the Appendix).

2.3.3. Reconfiguration.

In this section, we consider 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 2), 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 (as defined in the previous section). The reconfiguration procedure consists of the following steps:

1. Each follower continues to execute until it enters a deadlock condition.

2. All projection queues are cleared to prepare the replica for resuming the execution. After reaching deadlock, the remaining entries in the projection queues indicate a sequence of mutex acquisitions that is incompatible with the replicated application's algorithm (note that mutexes already acquired by the followers are valid) and, hence, must be removed.

---

[7] In 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.

3. Each follower chooses the new leader from the group membership list. It is assumed that all replicas contain identical lists so that a deterministic rule can be applied for the selection (e.g., pick the first replica in the list). If the follower is not chosen to be the new leader, then it waits in deadlock until the new leader starts sending mutex table updates. The new leader awakens all of its threads, allowing them to execute `lsa_lock` as the leader replica.

Note that if the leader-elect replica executes the above reconfiguration procedure faster than the other replicas, these replicas may receive mutex table updates from the new leader before they have reached a deadlock. It is necessary, therefore, to buffer mutex table updates in the followers after receiving the view change notification (i.e., after entering the reconfiguration mode). These buffered mutex table entries are transferred to the projection queues after step (2).

The reconfiguration algorithm presented above preserves correctness with respect to the new leader and any of the followers. Safety can be shown using a proof sketch similar to that for Theorem 1 (in the Appendix). Liveness is guaranteed by clearing the projection queues after reaching deadlock, thus allowing the followers to execute according to the mutex table updates received from the new leader.

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

In this section, we analyze 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 malicious leader can result in: (1) not sending a mutex table update at all, (2) sending a mutex table update only to some followers, or (3) sending a mutex table update with different contents (or in different orders) to different followers. These cases can result in the followers being inconsistent with each other. We sketch a solution to this problem that does not require the cost of a multicast protocol tolerating Byzantine failures. The approach we pursue takes action only after inconsistencies are detected by the voter, without incurring extra overhead during normal operation.

Failure Detection. The voter detects replica failures and, depending on the failure, decides upon system reconfiguration actions, as described below.

1. Detecting a follower crash or a spurious output from a follower 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 value error in a follower output 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 from the system. Note that in this case, because of the single-failure assumption, only the leader can crash.

Apart from the first case—in which no reconfiguration is needed—it is necessary to reconfigure the system and select a new leader among the nonfaulty followers that have not been excluded from the system.

Reconfiguration. To reconfigure the system, we select a subset of the remaining followers from which the system can restart through the following procedure: All followers send 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 are 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.

This can be done with the LSA algorithm since under the assumption that threads acquire mutexes infinitely often, the candidate moment is when all replica threads are suspended, which eventually happens after deadlock is reached.[8] Note that for a system with initially at least three replicas, a failure can degrade the system at most to a single running replica.

2.5. Performance Improvement

In the LSA algorithm as presented above, the transmission of a mutex table update from the leader to followers can happen only in two cases: when the table gets full, or periodically by `leader_periodic_tx` (if the table is not empty). Although this mechanism guarantees liveness, it may be not sufficient from the performance perspective, i.e., followers could spend a lot of their time waiting for mutex table updates to be received instead of doing useful computation. To overcome this problem, it is possible to introduce a follower-to-leader reliable unicast communication that can be used to signal the leader that a follower is willing to accept a new mutex table update (e.g., when a follower projection queue gets empty). In response to such a message from a follower, the leader may decide to multicast a new mutex table update to all followers.

To avoid that this new communication follower-to-leader could introduce an additional failure mode, the "flow-control" mechanism must be such that no follower can slow down (and possibly stop) the sending rate of mutex table updates. For example, if the leader keeps for each follower a predicate that is true when that follower is lacking mutex table updates, than the leader should multicast a mutex table update (in addition to the periodic and table-full mechanism) when any of these predicates is true.

Assuring that no follower can slow down the sending rate of mutex table updates can potentially allow a faulty follower to increase the sending rate without control, creating excessive traffic on the network and, hence, interfering with correct replicas. However, if mutex table updates are sent from the leader only when they are not empty, even if a faulty follower is requesting more mutex table updates than those that can be generated by the leader, no message is sent at all from the leader.

Finally, note that no mechanism has been considered to limit the lag between leader and followers. This has been done intentionally again to avoid the introduction of a failure mode due to the possibility of a follower (and hence a faulty follower) to slow down (and stop) the leader. In the LSA framework, this does not constitute a problem, as replicas are synchronized by the voter on blocking socket operations (e.g., when closing a client connection), which guarantees that the correct followers cannot lag behind the leader without limit.

2.6. Implementation Issues

Uniform naming convention. Because mutex table updates sent from the leader to followers contain information about threads and mutexes, it is necessary to have well-defined, replica-independent naming convention for them. In defining the logical ids for threads and mutexes, we assume that for all replicas, corresponding threads/mutexes are created/initialized by the same (logical) thread and in the same order in the context of this thread. For example, if thread t_A in replica A creates two child threads in the order t_{A_1} and t_{A_2}, then in replica B the thread t_B (corresponding to t_A) creates the child threads t_{B_1} and t_{B_2} in this same order. Threads t_{A_1} and t_{B_1} (t_{A_2} and t_{B_2}) need to perform the same computation (are corresponding).

A hierarchical naming scheme is employed in which a logical id of a thread is recursively defined:

```
logical_thread_id = parent_logical_thread_id ^ thread_creation_counter
```

where `thread_creation_counter` is a counter owned by each thread and incremented each time the thread spawns a new child thread. For example, for a thread t_n created by a thread t_{n-1} as $k_n{}^{th}$ child, the logical thread id would be $< k_1, k_2, \ldots, k_n >$. The logical name for a mutex is given by:

```
logical_mutex_id = (logical_thread_id, mutex_creation_counter)
```

where `mutex_creation_counter` is a private counter owned by each replica thread. This counter is incremented at the time of the mutex initialization. For example, the logical id of the nth mutex created by a thread t_k is given by (`logical_thread_id`$_k$, n).

---

[8]Note that the condition of all threads being suspended must be checked both when a thread is going to be suspended and when a thread is about to terminate.
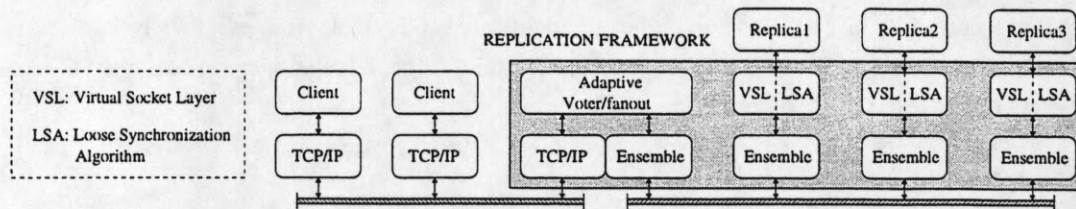
Figure 3. Replication framework.

Voter support. The LSA algorithm enforces determinism in the update of a given replica's state. Thread outputs from different replicas, however, can still be produced in different orders because of different thread scheduling. Therefore, before voting the voter must first group the replicas' outputs according to the logical thread id of the threads that generated them.

3. Application-Transparent Replication Framework

This section introduces a software framework consisting of an implementation of the loose synchronization algorithm, a virtual socket layer and a voter/fanout process for supporting the loose synchronization algorithm in replicating multithreaded applications. Figure 3 illustrates the configuration of a triplicated application employing this framework (Ensemble [14] is used as communication layer for reliable multicast protocol).

Multiple instances of an application execute on different nodes, and the clients have the illusion of a service that is implemented by a single, nonfaulty server. Multiple clients interact with the voter as if it were the real server. The voter forwards all data coming from clients to replicas using a FIFO-order reliable multicast protocol. Socket operations invoked by a replica's code (e.g., to send a response back to the client) are converted into requests that are sent to the voter/fanout process. The voter collects the requests for socket operations from all the replicas and, after voting, performs the actual operation on the physical socket associated with the specific client.

For a single-threaded application, voting requires replica *output consistency* (i.e., all correct replicas must send the same output messages to the voter in the same order). To guarantee replica output consistency, two conditions must be met: (1) *input consistency*, in which the input requests are identical and delivered to correct replicas in total order [10], and (2) *replica determinism*, in which, in the absence of faults, any execution of the replica starting from the same initial state and processing the same-ordered set of input requests leads to the same-ordered set of output messages [28].

For a multithreaded application, the voter groups the replica outputs on a per-thread basis. Therefore, *output consistency* and *input consistency* need to hold only with respect to corresponding threads across replicas. The condition of *replica determinism* is replaced with the condition that all replica threads are piecewise deterministic.

3.1. Loose Synchronization Algorithm

A prototype of the loose synchronization algorithm is implemented as a C/C++ library (LSA). The interception of lock operations on mutexes is performed transparently to the application by intercepting the application calls to the POSIX thread (PTHREAD) library. A set of macros is employed to override the PTHREAD functions at compile time. The inclusion of the LSA header file is the only change required to the application source code. Consequently, the application needs to be recompiled.[9]

3.2. Virtual Socket Layer

The virtual socket layer is compatible with the BSD socket interface and designed to hide the replication infrastructure from the replicated application using logical sockets instead of physical sockets. For instance, instead of calling the function `socket` to create a new socket, the replica calls `vsl_socket` instead. This function has the same signature (i.e., same input arguments

---

[9]The interception can also be done without requiring application recompilation by overriding the PTHREAD dynamic library.

11

and return type) as the corresponding BSD one, but returns a logical socket descriptor instead of a physical socket descriptor. The substitution can be easily automated with the help of preprocessor macros. For example, to instrument the *Apache 2.0.16* web server it was enough to insert the following two lines

```
#define   USE_VSL_ALIAS
#include  "vsl.h"
```

into 6 out of 447 C files and to insert the following initialization line

```
r_init(0, n_replicas);
```

into the file `server/main.c` which is a small effort for a code base of over 170,000 C source lines.

The virtual socket layer is responsible for (1) receiving/sending messages from the voter/fanout process and (2) interacting with the replicated application. A dedicated network thread in each replica accepts messages from the voter/fanout process. These messages correspond to new data and new connection requests arriving from clients, and they are buffered in a data queue or in a connection queue for the logical socket.

The concept of the virtual socket is similar to the idea of interposers proposed in the Eternal system [22]. The ORB invocations to the standard library for performing I/O are intercepted by Eternal and redirected to the Replication Manager, a process that conveys data from and to the replicas through the underlying reliable broadcast protocol. Involving the replication manager requires (1) an additional communication via named pipes, and (2) a context switch (because the replication manager is a separate process from the replica process). In our framework the virtual socket layer embeds the equivalent functionality of the Replication Manager into the replica process, eliminating the need for a separate process to redirect library invocations).[10]

3.3. Voter/Fanout Process

The virtual socket layer separates the replicated application from the voter, and the voter separates the replication infrastructure from the client. The voting mechanism is specific to the replicated application. A bit-wise comparison is a simple yet popular voting mechanism. Other alternatives (e.g., a check-sum verification or voting only on chunks of the data) can be incorporated in our voter implementation as well.[11]

While replicas use logical socket descriptors to interact with the virtual socket layer, the voter/fanout process uses real BSD sockets (*physical sockets*) and maps logical socket descriptors to physical socket descriptors. Socket operations are divided in two groups:

1. Operations that modify the physical socket state (e.g., `socket, bind, listen, close, connect, shutdown, send, write`) are voted upon by the voter. These operations correspond to socket operation requests sent from the replicas to the voter. Replicas can continue immediately after calling nonblocking functions (e.g., `send` and `write`). Blocking functions, however, do not return until the voter performs the function on the physical socket (e.g., `socket`).

2. Operations that do not modify the physical socket state (e.g., `accept, select, poll, recv, read`) are emulated by the virtual socket layer. The voter forwards data and client connection requests to the replicas for buffering. The virtual socket layer utilizes this buffered information when the replica invokes the emulated socket functions.

Note that the functions frequently executed are mostly either nonblocking or locally emulated by the virtual socket layer.

In addition to voting on outgoing messages, the voter/fanout process also forwards all client messages to the replicas using a FIFO-order reliable multicast protocol. The voter also provides (1) adaptive timeout estimation to minimize the probability of a false alarm when detecting hang errors, (2) timing error detection (the specifics of the replicated application can be embedded into the voter to override the adaptive timeout values, calculated statistically, with the maximum execution time allowed for the particular service request), and (3) *fast voting* (the voter can vote and send a response to the client as soon as the majority of replicas provide corresponding outputs that agree).

---

[10]In principle this same efficiency should be possible in Eternal as well.

[11]For example, an architecture for supporting voting in middleware is proposed in [5].

3.4. Adaptive Voter Timeout

To minimize the probability of false alarms and to reduce the latency in detecting crash and hung replicas, a mechanism for adapting the voter timeout is provided. The timeout value reflects: (1) the computation time required by the server to produce a response to the client and (2) the communication time.

Timeout detection. The voter enforces a timeout for each outstanding socket operation request from the replica. For each outstanding request, the voter maintains a separate timeout timer for each replica. These timers are used as follows to enforce the overall timeout for the request:

1. The initial value of each timer is set to the estimated mean time (see below) of the service time, plus a cushion factor of the standard deviation of the estimated mean time.

2. Start a timer upon receiving a new socket operation request from one of the replicas. The replica that issues the first request is called the *initiator*.

3. Declare that the initiator has failed if none of the other replicas sent the same request before the timer expired. Since there was only one request, it is reasonable to conclude that the initiator behaved incorrectly in making the request.

4. Declare that a replica has failed if (1) it does not generate the same request as the initiator before the timer expires, and (2) other non-initiator replicas have already made the same request. The fact that a majority of replicas make a request indicates that the silent replica is in error.

An exponential back-off mechanism is used to adjust the timeout value. When a replica fails to respond within the timeout period, the timeout value associated with the replica is doubled and a threshold counter is incremented. A replica is declared as failed only when the counter reaches a predefined value.

Timeout estimation. A timestamp is added to messages sent by the voter to each replica. Outgoing messages from the replicas include the timestamp for their corresponding input message. The voter computes the instantaneous service time for messages received from the replicas by subtracting the message timestamp from the current, real time. The instantaneous values of service time are used to estimate mean and deviation of the service time. Smooth estimates of the mean and the standard deviation are ensured by employing a low pass filter to attenuate noise and irrelevant fluctuations, as proposed by Jacobson in [15].

The objective is to estimate the service time necessary for each replica for processing a client request and generating corresponding response. A timestamp is added to messages sent by the voter to each replica. The replicas extract the timestamp from a received message and store it in a variable `last_timestamp` corresponding to the logical socket connection from which the message arrived. Messages sent by replicas to the voter include the value of `last_timestamp` associated with the logical socket connections for which the messages are destined. On receiving a message from a replica, the voter computes the instantaneous service time, subtracting the timestamp (extracted from the replica messages) from the current real time. The instantaneous values of service time are used to estimate mean and deviation of the service time. Smooth estimates of the mean and the standard deviation are ensured by employing a low pass filter to attenuate noise and irrelevant fluctuations, as proposed by Jacobson in [15].

Evaluation of the adaptive timeout algorithm. To evaluate the efficiency of the adaptive timeout estimation algorithm presented above, we trace the round-trip-time and timeout estimates (calculated by the voter) for a triplicated multithreaded Apache web server. The experimental setup consists of two Ethernet 100 Mbps LANs, one connecting the client to the voter and the other connecting the voter to all replicas. To stress the algorithm (to create unbalanced workload both on replica nodes and on the local network between the voter and the replicas), one of the three replicas ($Replica_3$) is executed on a Pentium III

The computed *timeout* closely follows RTT_i (see interval [12.5, 13.75]) as long as RTT_i varies smoothly. Abrupt changes in RTT_i (see intervals [13.75, 14.25] and [14.75, 15]) correspond to larger variance and, hence, *timeout* increases.
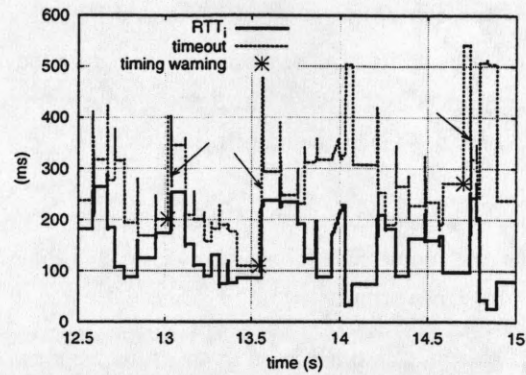
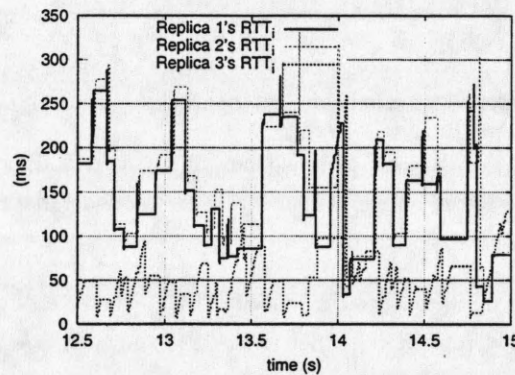**Figure 4. Timeout adaption for triplicated Apache ($Replica1$).**

Figure 5. Timeout adaption for triplicated Apache (all replicas).

800 MHz based machine, while the other two replicas ($Replica_1$ and $Replica_2$) and the voter are executed on a Pentium III 500 MHz. All machines run Linux 2.4, and Ensemble 1.20 [14] is used as networking layer providing the reliable broadcast protocol.

Figure 4 shows the instantaneous value of the round-trip-time (RTT_i) and the timeout value calculated (*timeout*) by the adaptive timeout algorithm for $Replica_1$.[12] One can see that the computed *timeout* closely follows the variations of RTT_i. Figure 4 also shows three cases of *timeout warning* generated by the voter to indicate that the replica did not send a message to the voter before the timeout associated with this replica expired. The timeout warning causes the timeout value to be doubled. On the second attempt the message from the replica is successfully delivered before the timer expires, and hence the *timeout* is recalculated and adapted to the replica RTT_i (see arrows in Fig 4).

Observe that the instantaneous values of RTT_i for $Replica_1$ range from about 30 ms to 300 ms (a factor of 10), and the algorithm is able to closely follow such variability. This illustrates the efficiency of an adaptive timeout estimation. Fixed value for the timeout would be either too small or too large.

Fig. (5) reports the instantaneous values of the round-trip-time for all three replicas. $Replica1$ and $Replica2$ have similar RTT_i, while $Replica3$ has a substantially smaller RTT_i (about 6 times). Recall that this replica executes on a faster machine.

4. Real-World Application: Apache Web Server

The Apache web server 2.0.16 was tested in an experimental setup consisting of two Ethernet 100 Mbps LANs, one connecting the client to the voter and the other connecting the voter to all replicas (see Figure 3). Replicas and voter execute on Pentium

---

[12]Mean and standard deviation estimates are not shown to keep the Fig 4 readable.

Table 5. Triplicated single-threaded Apache web server.

Experiment	Baseline (throughput)	TMR (throughput)	TMR (throughput reduction)
test.cgi	29.4 KBps	21.2 KBps	27.9 %
test2.cgi	114 KBps	86.7 KBps	23.9 %

III 500 MHz based machines. The client executes on a Pentium III 800 MHz based machine. All machines run Linux 2.4, and Ensemble 1.20 [14] is used as networking layer providing the reliable broadcast protocol. Apache was compiled with the *Multi-Processing Module with Threading via Pthread* (threaded) enabled. This module implements a hybrid multiprocess multithreaded server to handle client connections concurrently. Each process has a fixed number of threads.

In a broad sense, a client can utilize an Apache web server in two ways: (1) to retrieve static HTML pages (or files in general), and (2) to execute Common Gateway Interface (CGI) programs, which perform a computation and return a dynamic HTML page to the client. The first case is not interesting from an active replication point of view: since the page/file is static, a precalculated checksum can be added to the page/file so that errors in the retrieved data can be checked at the client side. The second case is interesting because checksum cannot protect the computation that produces the dynamic HTML page and this computation can be critical to the user.

If a multithreaded Apache/CGI system is to be replicated, then the LSA framework can be employed in the following manner: (1) In Apache, a mutex variable is used to serialize invocations to accept among several threads. accept returns the next available client connection, and the thread calling accept services this new connection. Instrumenting the accept mutex with the LSA algorithm guarantees that the same logical threads serve the same client connections in all replicas. (2) The LSA algorithm can be transparently used to ensure that corresponding CGI processes (i.e., processes created by the same logical thread in Apache) in each replica access shared values in the same order.

Experimental setup. To test Apache, a web benchmarking tool was used as a client application. One thousand client requests were sent (in groups of 10 simultaneous requests) for retrieving a dynamically generated HTML page. Two CGI programs were used to create variable load on the server and on the network: test.cgi generating a 123-byte HTML page, and test2.cgi representing a larger server load by generating a 1094-byte HTML page. The mechanism embedded in the voter for comparing outputs from replicas was adapted to skip replica/node-dependent fields in the messages generated by the Apache server[13] so that the voter would not raise false alarms.

Single-threaded Apache. The Apache server was initially instrumented only with the virtual socket layer (VSL) and was run in a single-threaded configuration (i.e., without using the LSA algorithm). This allowed us to measure the overhead due to the virtual socket layer plus the voter. Table 5 reports throughput and throughput reduction for the noninstrumented Apache (baseline) and the triplicated Apache, showing that throughput drops 24% when Apache is triplicated.

Multithreaded Apache. The next set of measurements was performed on Apache instrumented with the loose synchronization algorithm while varying the number of server threads. Each client request caused the server to acquire mutex variables seven times. One mutex access was used by Apache to serialize calls to the accept function. The other six acquisitions were used in the memory allocation routines (APR pools). These routines have no effect on the output seen by the voter (the HTML pages generated by Apache do not contain any references to local memory addresses). Thus, only the accept mutex needs to be instrumented to ensure output consistency. With this optimization, performance improved 2%. The experiment was repeated with the client requesting a static HTML page. In this case, the original Apache acquired mutex variables 207 times per client request, but only one access (to the accept mutex) was critical for output consistency. Instrumenting only the accept mutex

---

[13]For instance, the fields Date and Last-Modified contained in the HTTP OK message that precedes a HTML response (to a client request).
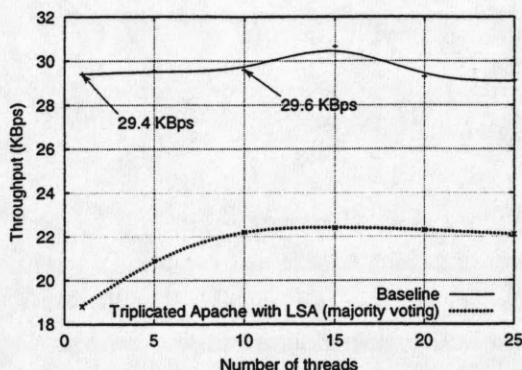
Figure 6. Triplicated multithreaded Apache under workload `test.cgi`.
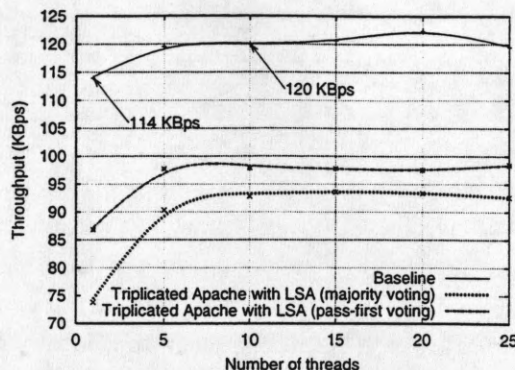
Figure 7. Triplicated multithreaded Apache under workload `test2.cgi`.

resulted in a 40% performance improvement. This indicates that the LSA algorithm can support a high mutex acquisition rate per client connection without incurring significant performance degradation.

Figure 6 compares the throughput as a function of the number of Apache threads under a light workload generated (`test.cgi`). Several conclusions can be drawn from this figure.

- The introduction of LSA algorithm to triplicated Apache with a single thread causes throughput to drop from 21.2 KBps (using a single-threaded, triplicated Apache without LSA algorithm) to 18.8 KBps (with LSA). This is due to the overhead of the LSA algorithm, which cannot be masked by exploiting concurrency.

- As the number of threads increases, additional concurrency can be exploited to improve throughput. For example, from Figure 6 with 10 threads, the throughput reaches 22.2 KBps (only a 25% throughput reduction as compared to the 29.6 KBps of the baseline configuration).

- Throughput does not increase further with more than 10 threads since client requests are sent in groups of 10 simultaneous requests.

Figure 7 presents the throughput as a function of the number of threads for the Apache server under a larger workload (`test2.cgi`). To further study the impact of the voter, we implemented a pass-first scheme in the voter (no message suppression was employed), which has lower overhead than the majority voting scheme (used in the experiments). The pass-first voting scheme causes throughput for 10 threads to increase from 92.8 KBps (22.7% throughput reduction as compared to 120 KBps of the baseline) to 97.9 KBps (18.4% throughput reduction).

Note that the experiments with `test2.cgi` show a higher throughput (for both the baseline and the instrumented multithreaded Apache) as compared to the experiments with `test.cgi`. For triplicated Apache with 10 threads, the throughput improvement (with respect to single-threaded, noninstrumented, triplicated Apache) is 4.7% for `test.cgi` and is 13% for `test2.cgi`.[14] Because the experiments were conducted on single-processor machines, multithreading allows an increase in throughput only when there is a computation time to be overlapped with I/O.

4.1. Discussion

The proposed replication framework consists of several software components, including the virtual socket layer, the voter/fanout process, the loose-synchronization algorithm, and Ensemble-based network communication layer. All these components contribute to the overall performance overhead. A set of measurements was conducted to quantify and to analyze the performance impact of entities constituting the replication framework and to compare the proposed framework with existing solutions.

---

[14]The change in throughput is calculated based on Figure 6 and Figure 7.
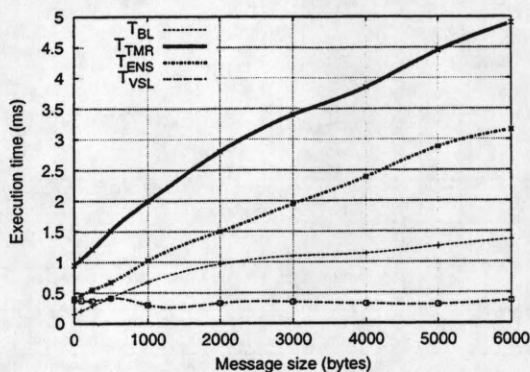
Figure 8. Execution time for a tripli-cated echo server broken down into three components.
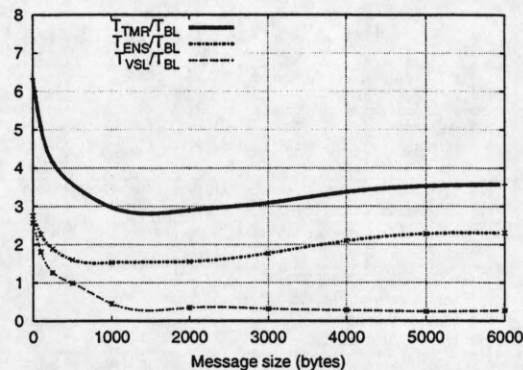
Figure 9. Ratio between execution times and T_{BL} for a triplicated echo server.

For comparing our framework with other replication system/frameworks Table 6 provides performance overhead reported for several existing approaches to replication. Table 6 clearly indicates that most replication strategies impose a significant performance overhead. The exception is Eternal with a reported overhead of 3%. Most of the overhead measurements [20], [22], [13] use an echo server (or effectively the same functionality).

An echo server accepts messages incoming from clients and echoes them back to the clients. The only computation performed by the server is moving the data received from the client to an internal buffer and sending the data stored in this buffer back to the clients. While the echo server barely resembles the kind of services that a real-world application would offer, we use it to facilitate comparison with existing studies (e.g., Eternal [22], AQuA [20]) where this simple application was employed in assessing performance. The data exchanged between the server and clients are immaterial for the correct functioning of the system. Also the communication is the dominant characteristic of the workload. Thus, issues such a value voting and maintaining significant application state become virtually irrelevant.

Performance measurements of the echo server replicated using the LSA framework. To characterize the overhead introduced by our replication framework, the performance of the triplicated echo server is compared with a simplex implementation of a TCP-based echo server (without any LSA code).[15] Figure 8 provides the execution time (as perceived by the client) of the baseline (T_{BL}) and triplicated echo server (T_{TMR}) as a function of the message size. For message sizes between 1K and 6K, the measured overhead (with respect to the execution time of the TCP-based implementation of the echo server) ranges from 200% to about 258% (see Figure 9).

To quantify the contribution to the overhead from different components of the replication framework, additional measurements were conducted and are also presented in Figure 8 and Figure 9. The results show that the Ensemble communication layer is a major source of the observed overhead (an average, T_{ENS} in Figure 8 is a 100% overhead for message sizes between 1K and 6K). Overhead introduced solely by our replication software (i.e., virtual socket layer and voter/fanout) decreases from 45% to 27% (with respect to T_{BL}) as message size grows from 1KB to 6KB (see T_{VSL}/T_{BL} in Figure 9).

Since LSA is implemented on top of TCP and Eternal in CORBA (which also uses TCP), it is nontrivial to make an accurate performance comparison. Ideally one would like to measure the execution times (or throughput) for the application in both frameworks under identical conditions. Since Eternal was not available, this was not a viable approach. Further, there are several architectural differences that make such comparison difficult. In the following we make an attempt to compare the two by normalizing both performance measurements with respect to TCP. The comparison is based on rough estimates and "back-of-the-envelope" calculations; nevertheless, it serves to provide a first order assessment. As the technologies, implementation,

---

[15]The measurements are conducted in the same testbed configuration as the one used for the experiments with the Apache web server.

Table 6. Overhead of replication frameworks.

Replication Framework	Application employed to measure performance	Overhead	Comments
AQuA [20]	CORBA-based echo server (VisiBroker). Active replication with pass-first scheme. Ensemble as a communication layer. Linux on Intel Pentium.	700%	Transparent replication to CORBA applications.
Eternal [22]	CORBA-based echo server (VisiBroker). Active replication with pass-first scheme. Totem as a communication layer. Linux on Intel Pentium.	3%	Transparent replication to both CORBA applications and ORB. 10% overhead on Solaris.
LSA framework	Active replicated echo server with majority voting. Ensemble as communication layer.	200%	30 % overhead for multithreaded Apache server with a CGI program creating a 1 KB HTML page.
Bast [13] OGS [13]	Smalltalk method invocation. CORBA method invocation. Solaris 2.5 on SPARC.	900% 900 %	Replicas immediately send a reply to the client.
Arjuna [24]	System call invocation. Solaris 2.3 on SPARC.	600% (for write system call)	Object oriented framework for fault tolerant distributed applications om transactional systems.
FATOMAS [25]	Agents that increment the value of a counter at each stage of the execution. AIX on PowerPC.	290%	Java-based fault-tolerant mobile agent system. Overhead calculated with respect to a triplicated agent v.s. single agent.

Table 7. TCP-based and CORBA-based simplex echo server.

message size (bytes)	TCP time (ms)	CORBA time (ms)	CORBA overhead (%)
1	0.136	0.552	306
100	0.171	0.591	246
1000	0.473	0.929	96.4
2000	0.684	1.20	75.4
4000	0.835	1.56	86.8
8000	1.19	2.41	103

and measurement of the performance overheads vary significantly, it is difficult to make a direct comparison among these measurements. However, since the Eternal reported overhead is so small, we investigated the potential reasons for the differences in the overhead we measured and the one observed in Eternal (keeping in mind the above caveat).

- *The overhead of CORBA with respect to plain TCP-based echo server.* An analysis shows that the overhead due to CORBA masks the overhead of the Eternal infrastructure itself. Recall that TCP version of the echo server is used as a baseline for comparison in our measurements. Table 7 reports the measurements of the overhead of a CORBA-based simplex echo server (VisiBroker, also used by Eternal) with respect to a TCP-based simplex echo server. One can see that the overhead varies from 306% to 75.4% for message sizes 1 byte and 2000 bytes, respectively. Consequently we estimate that the overhead incurred by Eternal-based echo sever will vary between 318% and 80.7% (depending on the message size) as compared with plain TCP-based echo server. The overhead is estimated using the relationship $\frac{T_{eternal}}{T_{tcp}} = \frac{T_{eternal}}{T_{corba}} \cdot \frac{T_{corba}}{T_{tcp}}$. The ratio $\frac{T_{eternal}}{T_{corba}}$ is 1.03 (3% overhead). The ratio $\frac{T_{corba}}{T_{tcp}}$ is calculated using data in Table 7.

- *Differences in the replication scheme.* The 3% overhead of Eternal is measured with respect to a replicated echo server with a pass-first scheme (no majority voting), where duplicated requests and responses are suppressed both at the sender and at the receiver. As a consequence, only two messages are sent: one for the request and one the response. Our framework employs majority voting, which involves exchanging of four messages: one is sent from the voter to the replicas and the remaining three come from replicas sending their outputs to the voter.

In addition to the pass-first scheme used in the previous Eternal experiment, Eternal also supports a majority voting scheme in which the voting occurs within the client process. The reported overhead for this scheme is 20-30% for triplicated client and triplicated server running a "packet driver" on Solaris [22]. In our framework, voting occurs outside the client in a separate voter/fanout process. As a result, there is an additional network communication, which adds an overhead of about 100% as compared with the unreplicated echo server (the communication is over plain TCP).

It should be emphasized that embedding the voter into the client body (as it is done in the Eternal system configuration) may be not sufficient in the case when the group of the replicated clients needs to produce a single output to the external world. In this scenario, the system will have to provide an additional voter to make a final decision as to the output that should be

Table 8. Error models used in injection experiments.

Error Model	Description	Failure Definition
SIGKILL	The OS delivers a SIGKILL signal to the target replica.	Target replica terminates, simulating a clean crash failure.
SIGSTOP	The OS delivers a SIGSTOP signal to the target replica.	Target replica threads are suspended, simulating a clean replica hang.
Text segment	A single bit in the text segment of the target replica is flipped	Target process can fail by crashing, hanging, or producing an incorrect state/output.
Heap memory	Bits in allocated regions of the heap memory in the target replica are periodically flipped until a failure is induced.	Target process can fail by crashing, hanging, or producing an incorrect state/output.

Table 9. SIGKILL, SIGSTOP, text-segment and heap injection results.

Error Model	Total Injected Errors	Activated Errors	Manifested Errors				
			CRASH			HANG	ASSERTION DETECTION
			SEG. FAULT	ILL. INSTR.	KILLED		
SIGKILL	200	200 (100%)	N/A	N/A	200 (100%)	N/A	N/A
SIGSTOP	200	200 (100%)	N/A	N/A	N/A	200 (100%)	N/A
Text segment	342	160 (47%)	111 (32%)	13 (4%)	N/A	14 (4%)	20 (6%)
Heap memory	214	N/A	20 (9%)	N/A	N/A	8 (4%)	106 (49%)

delivered to the outside, and this will add extra overhead similar to what we have measured. An alternative would be to replace the replicated client with a single client with an embedded voter. This, however, creates a risk of simultaneous failure of the client and the voter. The separation of the voter and the client, on the other hand, allows recovering from voter failures to be independent and possibly transparent to the client and the server.

5. Fault Injection Evaluation

A series of fault injection experiments were conducted to (1) assess the impact on application behavior of faults in the replicated application, the replication framework, and in Ensemble (the underlying reliable broadcast layer). A triplicated, multithreaded Apache server was used as the target application.[16] Table 8 summarizes the error models and the failure definition for each model. NFTAPE [30], a software framework for conducting automated error injection experiments, was used to conduct the experimental evaluation of the LSA algorithm.

Table 9 reports the results of over 1000 injections for all error models listed in Table 8. In all cases, the system is able to recover from a failure generated by the injection. Note that if the failed replica is the leader, followers successfully elected a new leader (after the failed leader was excluded from the system). Approximately 49% (106 runs) of heap injections were detected by assertions incorporated into the LSA code. Errors detected by assertions were caused by: (1) a corrupted entry in mutex table (98 cases), (2) a corrupted entry in the table mapping physical mutexes into logical mutex ids (6 cases), (3) a corrupted Ensemble data structure (1 case), and (4) invalid header in the synchronization messages (1 case).[17]

6. Related Work

In software-based replication, reliable message delivery and consistency of information constitute two major difficulties to overcome in the implementation of replicated systems. These issues have been extensively investigated and resulted in many group communication protocols [6], [14], [18], [3], [27]. Another fundamental issue in replicated systems is the potential nondeterminism in the execution of different instances of a replicated component/application.

The Tandem *Integrity S2* system [16] guarantees that its three processors execute the same instruction streams by synchronizing (1) on global memory accesses, (2) on hardware interrupts, and (3) periodically (every 4096 run cycles). The early work on software-based replication essentially emulated hardware solutions. For example, there are a number of systems in which replicas are synchronized at the interrupt level. The *TARGON/32* system uses a process-pair scheme with a LAN of three-processor machines connected by a dedicated bus for efficient reliable communication. Asynchronous events (e.g., UNIX

---

[16]No faults were injected into the voter.

[17]Out of 160 activated text segment errors, two errors did not manifest (i.e., a corrupted instruction was executed without having a visible impact on the behavior of the replica).

signals) are transformed into synchronous messages delivered to the destination process and its backup [1]. The *Hypervisor* system provides primary/backup replication transparently to the operating system and user applications. A virtual machine layer, inserted beneath the operating system, uses the hardware instruction counter [9] to count the instructions executed between two hardware interrupts. This information is collected on the primary machine and periodically sent over the network to the backup machine, which reproduces the effects of the primary's hardware interrupts [8]. *Transparent Fault Tolerance (TFT)* is similar to the Hypervisor solution, except the interpositioning is done at the operating system call interface [7].

Delta-4 provides user applications with passive replication, semiactive replication, and active replication. Active replication cannot handle nondeterminism of the replicas. In semiactive replication, a leader/follower model plus a preemption synchronization mechanism are employed. Replicas can be preempted only on a predefined set of *preemption points*. Each time an input message that requires preemption arrives at the leader, the leader determines the next preemption point on which the message will be served. This information is sent to followers so that they can serve the message at the same point as the leader [19], [23].

Synchronizing at the interrupt level in software causes large performance overhead, as synchronization information is transferred over a network. More recent software approaches to replication attempt to take advantage of the object-oriented paradigm and advocate object replication rather then process replication (as discussed above). A common trend in object replication has been to integrate fault tolerance via replication of CORBA applications [20].

Recent years have brought studies on replicating multithreaded applications. Some of the issues related to handling nondeterminism due to multithreading have been studied in the context of log-based rollback recovery. [2] suggests adding support to the Mach operating system to track and to log the order in which threads access locks and semaphores. The data preserved in the log is used to support rollback recovery of failed processes (i.e., the thread execution is replayed following the order dictated by the log). [29] presents a technique based on using a software counter to track the number of instructions between nondeterministic events during normal operation. In case of a failure, the instruction counts are used to force the replay of these events at the same execution points.

Existing solutions to replicate multithreaded applications are based on employing a nonpreemptive deterministic scheduler that guarantees the same scheduling on all replicas. *Eternal* addresses the replication of multithreaded CORBA objects by employing a nonpreemptive deterministic scheduler that allows the execution of only one logical thread at a time. As a result, concurrency is significantly limited. If the running thread executes a remote method invocation, for example, no other thread can be scheduled until the method returns and the running thread terminates processing [21].

Transactional Drago employs a deterministic, nonpreemptive scheduler to enforce deterministic behavior of multithreaded replicas. The algorithm 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*, such as a service request, selective reception, lock request, server call, or end of execution. Unlike Eternal, Transactional Drago allows the execution of more than one logical thread at a time; however, both can schedule only one physical thread at a time (even if multiple CPUs are available). As a result, Transactional Drago suffers limitations similar to Eternal's [26].

7. Conclusions

This paper proposed a *loose synchronization* algorithm for software-based active replication of multithreaded processes. The algorithm enforces "equivalent" ordering of state changes across replicas and guarantees replica consistency with low overhead. The leader replica establishes the order of mutex acquisitions and sends this order to the follower replicas over the network. The algorithm is formally specified and the proposed formalism is used to prove correctness of the algorithm in failure-free behavior as well as in presence of errors. To evaluate the proposed algorithm, a transparent active replication framework has been developed and used to replicate the multithreaded version of the Apache web server.

8. Appendix

8.1. Leader–Follower Correctness

This section outlines proofs of lemmas and theorems used in the main part of the paper. Based on the definitions and the LSA specifications (see the pseudocode in Figure 2) we prove the correctness of the LSA (safety and liveness properties). We start by proving two lemmas showing that the order in which conflicting mutex acquisitions are performed at followers agrees with the leader's history.

Lemma 1 *For each nonfaulty follower f, conflicting mutex acquisitions are ordered in H^f in the same way they are in $H^{l,f}$:*
$$\forall (m, t_i), (m, t_j) \in H^f, H^{l,f} : (m, t_i) \overset{H^f}{<} (m, t_j) \iff (m, t_i) \overset{H^{l,f}}{<} (m, t_j).$$

Proof. Note from the pseudocode that a thread t_i requesting a mutex m returns from `lsa_lock` (which means $(m, t_i) \in H^f$) only after extracting (m, t_i) from `proj_queue[m]`. Entries are not extracted from projection queues unless the mutex m has been locked. After extraction, `lsa_lock` never unlocks m; hence, extraction and exiting from `lsa_lock` (acquiring mutex m) are atomic. Therefore, $(m, t_i) \overset{H^f}{<} (m, t_j)$ if and only if (m, t_i) has been extracted from `proj_queue[m]` before (m, t_j). Since all conflicting operations on mutex m are appended to the bottom of `proj_queue[m]` by `on_mt_update` in the same order as they appear in $H^{l,f}$, (m, t_i) is extracted from `proj_queue[m]` before (m, t_j) if and only if $(m, t_i) \overset{H^{l,f}}{<} (m, t_j)$. $\square$

In absence of failures, the leader history reconstructed at a follower is a prefix of the actual leader history; hence, the previous lemma can be strengthen as follows.

Lemma 2 *Given the leader l and a follower f, in absence of failures conflicting mutex acquisitions are ordered in H^f in the same way they are in H^l: $\forall (m, t_i), (m, t_j) \in H^f, H^l : (m, t_i) \overset{H^f}{<} (m, t_j) \iff (m, t_i) \overset{H^l}{<} (m, t_j).$*

Based on Lemma 2 and the notion of causal dependency, we can now show that the leader and a follower satisfy the safety property.

Theorem 1 (Leader–Follower Safety) *Let H^l be the leader's history and H^f a follower's history. In absence of failures, H^l and H^f satisfy the safety property: $\forall (m, t) \in H^l, H^f : \theta_{H^l}(m, t) = \theta_{H^f}(m, t).$*

Proof. By contradiction, suppose that (m, t) is the first entry both in H^l and H^f such that $\theta_H(m, t) \neq \theta_{H'}(m, t)$. Therefore, there exists an earlier mutex acquisition (m'', t'') such that $(m'', t'') \in \theta_{H^l}(m, t) \wedge (m'', t'') \notin \theta_{H^f}(m, t)$ or $(m'', t'') \notin \theta_{H^l}(m, t) \wedge (m'', t'') \in \theta_{H^f}(m, t)$. By definition of causal precedence, there are three cases to consider.

Case (1): $t'' = t$. The condition on (m'', t'') corresponds either to $(m'', t) \overset{H^l}{<} (m, t) \wedge (m, t) \overset{H^f}{<} (m'', t)$ or to $(m, t) \overset{H^l}{<} (m'', t) \wedge (m'', t) \overset{H^f}{<} (m, t)$. This means that the behavior of the leader thread t and the follower thread t have diverged. If there is no mutex acquisition common to both, then the threads' behaviors have diverged before they required their first mutex. However, this contradicts the piecewise thread determinism assumption since leader and follower replica start from the same initial state. On the contrary, let (m^*, t) be the last mutex acquisition common to both leader and follower thread t. After that acquisition, the two threads acquire different mutexes. However, note that by construction $(m^*, t) < (m, t)$ both in H^l and H^f; thus, by definition of (m, t), $\theta_{H^l}(m^*, t) = \theta_{H^f}(m^*, t)$, which—for the piecewise thread determinism assumption—contradicts the fact that leader and follower thread t behaviors have diverged after (m^*, t).

Case (2): $m'' = m$. The condition on (m'', t'') corresponds either to $(m, t'') \overset{H^l}{<} (m, t) \wedge (m, t) \overset{H^f}{<} (m, t'')$ or to $(m, t) \overset{H^l}{<} (m, t'') \wedge (m, t'') \overset{H^f}{<} (m, t)$. This contradicts Lemma 2.

Case (3): $t'' \neq t \wedge m'' \neq m$. There must be a chain of causal dependencies from (m'', t'') to (m, t). Note that when going from one element of this chain to the following element, it is not possible that both the thread and the mutex change (see definition of causal dependency). Let (m^*, t^*) be the element in this chain that immediately precedes (m, t), i.e., $(m'', t'') \rightsquigarrow$

$\cdots \rightsquigarrow (m^*, t^*) \rightsquigarrow (m, t)$. For (m^*, t^*) to exist, it must be that either $t^* = t$ or $m^* = m$. So, the existence of (m^*, t^*) leads to one of the two conditradictions above; therefore, such a chain from (m'', t'') to (m, t) cannot exist. □

The following lemma is used to prove liveness between the leader and a follower.

Lemma 3 *In absence of failures, if (m, t) exists in the leader's history, then it will be eventually appended to* `proj_queue[m]` *in each follower.*

Proof. In absence of failures, if $(m, t) \in H^l$, then it has been stored in the leader's mutex table, which is eventually transmitted to the followers (either when it gets full or by `leader_periodic_tx`). Hence, eventually (m, t) is appended to `proj_queue[m]` by `on_mt_update`. □

Theorem 2 (Leader–Follower Liveness) *In absence of failures, the following two conditions hold.*

1. *If a mutex acquisition is performed at the leader l, eventually it will be performed at each follower f:* $(m, t) \in H^l \implies \Diamond (m, t) \in H^f$.

2. *If a mutex acquisition is performed at a follower f, eventually it will be performed at the leader l:* $(m, t) \in H^f \implies \Diamond (m, t) \in H^l$.

Proof. Consider condition (1). Note that when (m, t) is extracted from `proj_queue[m]`, t returns from `lsa_lock`, which is equivalent to $(m, t) \in H^f$. The proof is by induction on the position n of (m, t) in the sequence H^l.

Base case. Assume $n = 1$ (i.e., (m, t) is the first element of H^l). We first show that since thread t is in the leader, eventually it will be in the follower as well. If t is the replica's main thread, it is present in both leader and follower. If t is not the main thread, then t has been be created in the leader by a thread t_p. For the picewise thread determinism and same initial state assumptions, if t_p is created in the follower, then eventually t will be created in the follower as well. If t_p is not the replica's main thread, the same argument can be iterated to show that, in fact, t will be eventually created in the follower.

Note that $index(m, t) = 1$, since (m, t) is the first mutex acquired at the leader and, so, the first mutex acquired by the leader thread t. The first mutex the follower thread t will request is also m for the piecewise determinism and same initial state assumptions. If the follower thread t tries to acquire m (i.e., invokes `lsa_lock(m)`) before (m, t) is appended to `proj_queue[m]`, then the condition $can\_acquire\_mutex(m)$ at line 13 will be *false* because `proj_queue[m]` is initially empty. Therefore, t is suspended (line 26). For the Lemma 3, eventually `on_mt_update` will append (m, t) to `proj_queue[m]`, which becomes the top entry. `on_mt_update` will also resume t since the condition $can\_schedule\_next\_thread(m)$ will be true. Once resumed, t will find the condition $can\_acquire\_mutex(m)$ *true* and so it will remove (m, t) at line 13. If t tries to acquire m after (m, t) is appended to `proj_queue[m]`, then it will find the condition $can\_acquire\_mutex(m)$ *true* and so it will remove (m, t) at line 14.

Inductive step. Assume the theorem is true for n and that (m, t) is the $(n + 1)^{th}$ element in H^l. We first show that since thread t is in the leader, eventually it will be in the follower as well.

If (m, t) is not the first mutex acquisition of thread t, then t is already present in the follower, otherwise we need to show that eventually t will be created in the follower. If t is the replica's main thread, it is present in both leader and follower. If t is not the main thread, then t has been be created in the leader by a thread t_p. If t_p does not acquire any mutex before creating t, for the picewise thread determinism and same initial state assumptions, if t_p is created in the follower, then eventually t will be created in the follower as well. If t_p acquires a mutex before creating t, consider the mutex acquisition (m', t_p) of t_p immediatelly preceding the creation of t. Since this acquisition precedes (m, t) in H^l, by hypothesis, eventually it must in H^f as well. For the safety property $\theta_{H^l}(m', t_p) = \theta_{H^f}(m', t_p)$; hence, for the picewise thread determinism and same initial state assumptions, if t_p is created in the follower, then eventually t will be created in the follower as well. If t_p is the replica's main thread, it is present in both leader and follower; otherwise, the same argument can be iterated to show that, in fact, t will be eventually created in the follower.

Assume and $index(m,t) = k+1$. By hypothesis, all t's mutex acquisitions (m_i, t) contained in the first n positions of H^l eventually are in H^f (t eventually acquires the mutexes m_i through lsa_lock). The number of such acquisitions must be k because $index(m,t) = k+1$. Moreover, since no follower thread except t can remove entries of the form (m_i, t) from proj_queue[m_i], all these k acquisitions must be done by t. Let (m_k, t) be the last of these acquisitions. For the safety property, $\theta_{H^l}(m_k, t) = \theta_{H^f}(m_k, t)$; hence, for the piecewise thread determinism and same initial state assumptions, the $(k+1)^{th}$ mutex the follower thread t will request is m (the same as the leader thread t). Consider now two cases.

(a) Assume that there is no (conflicting) acquisition (m, t_i) preceding (m, t) in the projection to m of the first n elements of H^l. If t requests m before (m, t) is appended in proj_queue[m], then t is suspended. When t is resumed by on_mt_update, it will find the condition $can\_acquire\_mutex(m)$ *true* and so it will remove (m, t) at line 14. If t requests m after (m, t) is appended to proj_queue[m], then it will find the condition $can\_acquire\_mutex(m)$ *true* and so it will remove (m, t) at line 14.

(b) Assume that there is a (conflicting) acquisition preceding (m, t) in the projection to m of the first n elements of H^l and let (m, t_i) be the one immediately preceding (m, t). If thread t tries to lock m before (m, t_i) is extracted or (m, t) is appended to proj_queue[m], then t is suspended (it will find at line 13 the condition $can\_acquire\_mutex(m)$ *false*). t will be resumed when (m, t) becomes the top entry in proj_queue[m]. This happens either at line 14, when (m, t_i) is eventually removed[18], or if proj_queue[m] is empty when (m, t) is appended to it by on_mt_update. Once resumed, t will find the condition $can\_acquire\_mutex(m)$ *true* and so it will remove (m, t) at line 14. If t tries to lock m after (m, t_i) is extracted and (m, t) is appended to proj_queue[m], then it will find the condition $can\_acquire\_mutex(m)$ *true* and so it will remove (m, t) at line 14.

Consider condition (2). The conclusion follows from the fact that $(m, t) \in H^f \implies (m, t) \in H^{l,f}$ (i.e., only mutex acquisitions that are in $H^{l,f}$ can be granted) and from the fact that $H^{l,f}$ is a prefix of H^l. □

Theorem 3 (Leader–Follower Correctness) *In absence of failures, the leader l and a follower f satisfy the correctness property.*

8.2. Follower–Follower Correctness

Safety, liveness (and, hence, correctness) between two nonfaulty followers are shown under the assumption that they receive the same sequence of leader's messages (which implies that leader's histories reconstructed by them are such that one is prefix of the other). This condition always holds in the failure-free scenario. More importantly, the condition also holds when a corrupted mutex table is sent to the followers as long as all followers receive the same mutex table.

Lemma 4 *Given two nonfaulty followers f_1 and f_2 with H^{l,f_1} and H^{l,f_2} such that one is prefix of the other, conflicting mutex acquisitions are ordered in H^{f_1} in the same way they are in H^{f_2}:* $\forall (m, t_i), (m, t_j) \in H^{f_1}, \in H^{f_2} : (m, t_i) \overset{H^{f_1}}{<} (m, t_j) \iff (m, t_i) \overset{H^{f_2}}{<} (m, t_j)$.

Proof. Let (m, t_i) and (m, t_j) be both in H^{f_1} and H^{f_2}. Based on the Lemma 1 and on the fact that H^{l,f_1} and H^{l,f_2} are such that one is prefix of the other, it straightforward to show that (m, t_i) and (m, t_j) must be both in H^{l,f_1} and H^{l,f_2}. □

Theorem 4 (Follower–Follower Safety) *Given two nonfaulty followers f_1 and f_2 with H^{l,f_1} and H^{l,f_2} such that one is prefix of the other, they satisfies the safety property:* $\forall (m, t) \in H^{f_1}, H^{f_2} : \theta_{H^{f_1}}(m, t) = \theta_{H^{f_2}}(m, t)$.

Proof Sketch. The proof follows the same steps as for Theorem 1 except that instead of using Lemma 2, it is necessary to use Lemma 4. □

---

[18]Note that $(m, t_i) \overset{H^l}{<} (m, t)$; hence, it will be eventually removed by hypothesis.

Lemma 5 *Given two nonfaulty followers f_1 and f_2 that receive the same sequence of leader's messages, if (m, t) exists in H^{f_1} then it will be eventually appended to* proj_queue[m] *of f_2.*

Proof. If $(m, t) \in H^{f_1}$, then $(m, t) \in H^{l, f_1}$. If (m, t) is not already in H^{l, f_2}, it will eventually be because f_1 and f_2 receive the same leader's messages. Once in H^{l, f_2}, (m, t) will be appended to proj_queue[m] of f_2 by on_mt_update. $\square$

Theorem 5 (Follower–Follower Liveness) *Given two nonfaulty followers f_1 and f_2 that receive the same sequence of leader's messages, they satisfies the liveness property:* $(m, t) \in H^{f_1} \implies \Diamond(m, t) \in H^{f_2}$.

Proof Sketch. The proof follows the same steps as for Theorem 2 except that instead of using Lemma 3, it is necessary to use Lemma 5. $\square$

Note that while liveness between leader–follower guarantees a continuous operation, liveness between follower–follower only guarantees that followers will eventually grant the same mutex acquisitions.

Theorem 6 (Follower–Follower Correctness) *Given two nonfaulty followers f_1 and f_2 that receive the same sequence of leader's messages, they satisfies the correctness property.*

8.3. Deadlock

Theorem 7 *A nonfaulty follower is in deadlock if and only if the following condition holds:* $\forall m \in mutexes : (\Box proj\_queue[m] = \emptyset) \vee (proj\_queue[m].head().t \in suspended\_threads) \vee (\Box proj\_queue[m].head().t \notin threads).$[19]

Proof Sketch. $\Rightarrow$: Suppose there exists a mutex m that violates the above condition. If the thread $proj\_queue[m].head().t$ (which is a valid thread and is not suspended) requests mutex m, then the request can be served, contradicting the assumption of deadlock.

$\Leftarrow$: From the hypothesis, projection queues can be partitioned in three classes: (1) those the are and will always be empty, (2) those whose thread in the top entry is suspended, and (3) those whose thread in the top entry does not and will always not exist. Mutexes corresponding to class (1) and (2) will never be acquired because no top entry can be removed. Threads in top entries of class (2) either form cyclic dependencies or depend on a mutex corresponding to class (1) or (3); therefore, no mutex can be acquired. $\square$

The following theorem exploits the presence the artificial mutex mc to express deadlock with a simpler condition than that of Theorem 7. This new condition is used by the LSA pseudocode (predicate *deadlock*) for detecting deadlock during reconfiguration.

Theorem 8 *A nonfaulty follower is in deadlock if and only if the following condition holds:* $\forall m \in mutexes : (\Box proj\_queue[m] = \emptyset) \vee (proj\_queue[m].head().t \in suspended\_threads) \vee (proj\_queue[m].head().t \notin threads).$

Proof. $\Rightarrow$: We first show that if the replica is in deadlock then $\forall m \in mutexes : \Box((proj\_queue[m] = \emptyset) \vee (proj\_queue[m].head().t \in suspended\_threads) \vee (proj\_queue[m].head().t \notin threads))$. Supposing by contradiction that this condition does not hold, i.e., $\exists m \in mutexes : \Diamond((proj\_queue[m] \neq \emptyset) \wedge (proj\_queue[m].head().t \notin suspended\_threads) \vee (proj\_queue[m].head().t \in threads))$, then eventually the thread $proj\_queue[m].head().t$ will exist, will not be suspended, and will be in the top entry of $proj\_queue[m]$. Therefore, if this thread requests m, the request will be eventually served, contradicting the hypothesis of deadlock.

Since $\Box((proj\_queue[m] = \emptyset) \vee (proj\_queue[m].head().t \in suspended\_threads) \vee (proj\_queue[m].head().t \notin threads))$ is equivalent to $(\Box proj\_queue[m] = \emptyset) \vee (\Box proj\_queue[m].head().t \in suspended\_threads) \vee (\Box proj\_queue[m].head().t \notin threads)$, and since if a predicate is true from now on, it is also true now, it follows that $\forall m \in mutexes : (\Box proj\_queue[m] = \emptyset) \vee (proj\_queue[m].head().t \in suspended\_threads) \vee (proj\_queue[m].head().t \notin threads)$.

---

[19]We use the linear temporal logic symbol $\Box$ to denote *from now on*.

$\Leftarrow$: From the hypothesis, projection queues can be partitioned in three classes: (1) those the are and will be empty, (2) those whose thread in the top entry is suspended, and (3) those whose thread in the top entry is not in the set of the current threads. Clearly, mutexes corresponding to class (1) will never be acquired since no more entries will be stored in any projection queue.

Mutexes corresponding to class (3) can be acquired if and only if the thread in their top entry is created in the future. But for this the mutex mc—which is used to serialize accesses to the function create_new_thread—must be acquired by the parent. If mc corresponds to class (3), then it is evident that no thread can be created in the future and, thus, mutexes corresponding to class (3) will never be acquired. Therefore, assume that mc corresponds to class (2).

Mutexes corresponding to class (2) can be acquired in the future if and only if the thread in the top entry of their projection queue is resumed. But these threads are suspended because are waiting for the top entry of another projection queue to be removed, a projection queue that must be either in class (1), (2), or (3) by hypothesis. These dependencies imply that no thread can be awaken at all; hence, no mutex can be acquired (and in particular mc). $\square$

The following corollary is a consequence of the follower–follower correctness theorem.

Corollary 1 *Given two nonfaulty followers f_1 and f_2, if one deadlocks, eventually so does the other: f_1 deadlocks $\implies$ $\Diamond f_2$ deadlocks.*

Proof. By contradiction, suppose there exists (m, t) that is acquired in f_2 (i.e., $(m, t) \in H^{f_2}$) but cannot be acquired in f_1 because f_1 is in deadlock. Since f_1 and f_2 satisfy the liveness property (in both directions; hence, in the $f_2 \to f_1$ direction), eventually (m, t) must be in H^{f_1} (i.e., will be acquired in f_1), which contradicts the hypothesis. $\square$

Deadlock detection during normal operation. The absence of output observed by the voter (replica hang) may be caused by a replica thread being suspended because of a deadlock condition on a subset of the replica mutexes. The generalization of Theorem 7 for a subset of mutexes $\mathcal{M}'$ is given by: (A) $\forall m \in \mathcal{M}' : (\Box proj\_queue[m] = \emptyset) \lor (proj\_queue[m].head().t \in suspended\_threads \land (\exists m' \in \mathcal{M}' : proj\_queue[m].head().t\ waits\_for\ m')) \lor (\Box proj\_queue[m].head().t \notin threads)$, where the predicate $t\ waits\_for\ m'$ is true if the thread t is suspended because of requesting mutex m'. The difference between (A) and Theorem 7 is in an extra predicate in the second condition, which accounts for $\mathcal{M}'$ possibly not containing all the replica mutexes by requiring that dependencies remain in $\mathcal{M}'$. The deadlock condition on a subset of mutexes corresponds to the existence of a subset $\mathcal{M}'$ for which (A) holds.

In practice, the $\Box$ operator preceding a predicate p can be evaluated by observing that p has held for a sufficient amount of time T, where T is the maximum waiting time on a projection queue and includes the maximum time between reception of subsequent mutex table updates.

The existence of a subset $\mathcal{M}'$ for which (A) holds can be evaluated with linear complexity by forming a dependency graph. Nodes are both the threads in the top entries that are suspended—the first part of second condition of (A)—and mutexes for which either the projection queue is empty or the thread in the top entry is not valid—the first and third conditions of (A). An edge connects a thread t_1 to a thread t_2 if t_1 is suspended because of requesting the mutex for which t_2 is top entry; an edge also connects a thread t to a mutex m if t is suspended because of requesting m—the second part of second condition of (A). A subset $\mathcal{M}'$ for which (A) holds exists if there is a cycle in the graph or there is a path terminating on a mutex whose status has not changed for more than T time units since the last time a thread was suspended because of requesting this mutex.

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] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Proceedings of the 22nd International Symposium on Fault Tolerant Computing*, pages 76–84, 1992.

[4] O. Babaoglu and K. Marzullo. *Distributed Systems*, chapter Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms, pages 55–96. Addison-Wesley, 1993.

[5] D. Bakken, Z. Zhan, C. Jones, and D. Karr. Middleware support for voting and data fusion. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 453–463, 2001.

[6] K. Birman and R. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[7] T. C. Bressoud. TFT: A software system for application-transparent fault tolerance. In *Proceedings of the 28th International Symposium on Fault Tolerant Computing*, pages 128–137, 1998.

[8] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, 1996.

[9] T. Cargill and B. Locanthi. Cheap hardware support for software debugging and profiling. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 82–83, 1987.

[10] F. Cristian, H. Aghali, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118(1):158–179, 1995.

[11] E. Elnozahy, D. Johnson, and Y. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical report, Carnegie Mellon University, 1996.

[12] P. D. Ezhilchelvan, R. A. Macedo, and S. K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 296–306, 1995.

[13] R. Guerraoui, P. Felber, B. Garbinato, and K. R. Mazouni. System support for object groups. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 1998.

[14] M. Hayden. *The Ensemble System*. PhD thesis, Dept. of Computer Science, Cornell University, USA, 1997.

[15] V. Jacobson. Congestion avoidance and control. In *Proceedings of the ACM SIGCOMM '88 Conference*, pages 314–329, 1988.

[16] D. Jewett. Integrity S2: A fault-tolerant Unix platform. In *Proceedings 21st International Symposium on Fault-Tolerant Computing*, pages 512–519, 1991.

[17] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences*, volume 3, pages 317–326, 1998.

[18] L. E. Moser et al. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996.

[19] M. Chereque et al. Active replication in Delta-4. In *Proceedings of the 22nd International Symposium on Fault Tolerant Computing*, pages 28–27, 1992.

[20] M. Cukier et al. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the 17th Symposium on Reliable Distributed Systems*, pages 245–253, 1998.

[21] 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.

[22] P. Narasimhan. *Transparent Fault Tolerance for Corba*. PhD thesis, Dept. of Electrical and Computer Engineering, University of California, Santa Barbara, USA, 1999.

[23] P. A. Barrett et al. The Delta-4 extra performance architecture (XPA). In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing Systems*, pages 481–488, 1990.

[24] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The design and implementation of arjuna. *Computing Systems*, 8(2):255–308, 1995.

[25] S. Pleisch and A. Schiper. Fatomas - a fault-tolerant mobile agent system based on the agent-dependent approach. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 215–224, 2001.

[26] S. A. R. Jimenez-Peris, M. Patino-Martinez. Deterministic scheduling for transactional multithreaded replicas. In *Proceedings of the 19th Symposium on Reliable Distributed Systems*, 2000.

[27] M. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the Second ACM Conference on Computer and Communication Security*, pages 68–80, 1994.

[28] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[29] J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proceeding of the 26th International Symposium on Fault Tolerant Computing*, pages 250–259, 1996.

[30] D. Stott, B. Floering, Z. Kalbarczyk, and R. Iyer. Dependability assessment in distributed systems with lightweight fault injectors in nftape. In *Proceedings of the International Computer Performance and Dependability Symposium*, 2000.