

A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas

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

Abstract

Software-based active replication is expensive in terms of performance overhead. Multithreading can help improve performance; however, thread scheduling is a source of nondeterminism in replica behavior. This paper presents a Preemptive Deterministic Scheduling (PDS) algorithm for ensuring deterministic replica behavior while preserving concurrency. Threads are synchronized only on updates to the shared state. A replica execution is broken into a sequence of rounds, and in a round each thread can acquire up to two mutexes. When a new round fires, all threads' mutex requests are known; thus, it is possible to form a deterministic scheduling of mutex acquisitions in the round. No inter-replica communication is required. The algorithm is formally specified, and the proposed formalism is used to prove its correctness.

Failure behavior and performance of a PDS algorithm's implementation are evaluated in a triplicated system and compared with two existing solutions: nonpreemptive deterministic schedulers and the Loose Synchronization Algorithm (LSA) proposed by the authors in an earlier paper. The results show that PDS outperforms nonpreemptive deterministic schedulers. Compared with LSA, PDS has lower throughput; however, it provides additional benefits in terms of system dependability and, hence, can be considered as a trade-off between performance and dependability. These characteristics are investigated with fault injection.

1 Introduction

Software-based active replication is a well-known technique for providing fault tolerance using space redundancy and fault-masking. Typically, replication is expensive in terms of performance overhead [1–3]. Multithreading can help improve performance by exploiting concurrency in thread execution;¹ however, since the thread/process scheduling performed by operating systems such as UNIX is asynchronous with replica execution, multithreaded replicas can exhibit *nondeterministic* behavior.

Solutions to replicate multithreaded applications/objects are based on a nonpreemptive deterministic scheduler, which enforces the same thread interleaving on all replicas to achieve determinism in replica state updates. In *Eternal*, determinism is achieved by processing application threads (each of which serves a client request) sequentially, which is effectively a single-threaded solution [4]. In *Transactional Drago*, the executions of multiple logical threads are interleaved: if the running thread starts an I/O operation, the thread is suspended waiting for I/O to complete while another thread may be scheduled

¹Note that the performance overhead due to group communication is typically non-negligible and cannot be reduced by multithreading.

[5].² Nonpreemptive deterministic schedules, although providing consistent replica behavior, cannot exploit concurrency in thread execution, since only one physical thread is scheduled at a given time. This results in poor scalability and performance of the replicated system.

In contrast with nonpreemptive deterministic schedulers, [6] describes a *Loose Synchronization Algorithm (LSA)* to maintain multithreaded replica consistency. The algorithm enforces a compatible sequence of state updates in all replicas without requiring the same thread interleaving. This is achieved by intercepting mutex lock/unlock operations performed by application threads on accessing the shared data. Intercepting mutex lock/unlock operations was first suggested in [7] for message-logging-based recovery.

In the LSA algorithm, one replica (leader) decides the mutex acquisition order and propagates it to other replicas (followers), which enforce the leader-dictated order on the execution of their threads. While the method preserves a large degree of concurrency, the presence of inter-replica communication can affect overall system dependability (as shown in § 5.2).

This paper proposes a *Preemptive Deterministic Scheduling (PDS)* algorithm, with no leader/follower structure and no inter-replica communication. In PDS, the key mechanism to ensure determinism is the concept of a *round* (similar to the notion of a barrier in parallel computing). A replica's execution is broken into a sequence of rounds, and in a round each thread can acquire up to two mutexes. Any thread can trigger a new round. On requesting a mutex, a thread t checks if it can acquire the mutex m it requests. If t cannot acquire m (e.g., m is held by another thread or t has already acquired a maximum number of mutexes for that round), it then checks whether all other threads are suspended. If so, t starts a new round; otherwise, t is suspended. When a new round is started, all threads' mutex requests are known, and therefore, a deterministic scheduling of mutex acquisitions naturally occurs: threads simultaneously requesting the same mutex acquire it according to increasing thread ids.

While typically most algorithms for providing fault tolerance services are formally proved for correctness, this paper additionally advocates the use of error injection for sound assessment of a proposed algorithm's dependability. Employing error-injection allows us to study the failure behavior of the overall system (including the algorithm) under realistic scenarios, which cannot be achieved by using formal methods alone. The major contributions of this paper are:

- Specification, proof of correctness, and implementation of

²To guarantee determinism, there are situations in which the algorithm waits for the I/O to complete (keeping the CPU idle) although another thread can be scheduled.

a Preemptive Deterministic Scheduling (PDS) algorithm for maintaining multithreaded replica consistency;

- Study of performance-dependability trade-offs in selecting deterministic scheduling algorithms when replicating multithreaded applications. The target strategies include PDS, LSA, and a nonpreemptive deterministic scheduler (NPDS).

A performance evaluation shows that PDS and LSA outperform NPDS by providing, respectively, two and five times more throughput. This is because PDS and LSA can schedule multiple threads to execute at the same time.

An error-injection-based evaluation of dependability shows that because LSA relies on an inter-replica communication channel for efficient scheduling of mutex acquisitions, the algorithm is more sensitive to the underlying communication layer's fail silence violations. This leads to a larger number of catastrophic failures (i.e., cases in which the entire replicated system fails) for LSA than for PDS. (NPDS dependability characteristics are similar to those of PDS because neither of the two algorithms uses inter-replica communication.) Therefore, if minimizing downtime is crucial (as it is for highly available systems), PDS is a more appropriate choice than LSA; if performance concerns have priority over minimizing downtime, then LSA can be preferred to PDS.

Finally, error-injection experiments make it evident that errors originating from the underlying communication layer (Ensemble [8] in our experiments) do propagate and can lead to catastrophic failures of the entire replicated system. Consequently, we argue that a middleware that provides fault tolerance services (e.g., reliable communications, process recovery) to applications should itself be fault-tolerant.

2 Related Work

Early research on software-based replication focused on synchronizing replicas at the interrupt level. For example, in the *TARGON/32* system, asynchronous events (e.g., UNIX signals) are transformed into synchronous messages delivered to the destination process and its backup [9]. In the *Hypervisor* system (based on a primary/backup model) a virtual machine layer, beneath the operating system, uses a hardware register to count the instructions executed by a primary machine between two hardware interrupts [10]. This information is sent over the network to a backup machine. The backup uses instruction counts to reproduce the effects of the primary's hardware interrupts with respect to the backup's instruction stream. *Delta-4* provides semi-active replication with a leader/follower model and a preemption-synchronization mechanism. When an interrupt arrives, the leader determines the next preemption point at which the interrupt will be served and sends this information to followers. The scheme is called semi-active because only the leader interacts with the clients [11]. Synchronizing at the interrupt level in software suffers from large performance overhead due to the necessity of transferring fine granularity synchronization information over a network. In [12], nondeterminism is solved for real-time systems off-line, via schedulability analysis.

More recent software approaches to replication attempt to take advantage of the object-oriented paradigm and advocate object replication rather than process replication (as discussed above). *AQuA* provides transparent, single-threaded replication to CORBA objects by means of proxies [1].

Solutions to replicate multithreaded applications/objects are based on a nonpreemptive deterministic scheduler. In *Eternal*, application threads are processed sequentially [4]. In *Transactional Drago*, the executions of multiple logical threads are interleaved on performing I/O operations [5].

The *Loose Synchronization Algorithm* [6], proposed by the authors in an earlier paper, captures the natural concurrency in a leader replica and projects it on follower replicas through inter-replica communication. To the best of our knowledge, this is the first multithreaded solution for maintaining replica consistency.

3. System Model: Definitions and Assumptions

The system consists of a set of identical multithreaded processes (replicas) running on different nodes. Each replica consists of a set of threads \mathcal{T} and a set of mutexes \mathcal{M} used to protect partitions of shared data (\mathcal{T} and \mathcal{M} can be infinite). It is assumed that the same thread/mutex ids are associated with corresponding threads/mutexes of different replicas; this can be enforced by using a hierarchical thread/mutex naming scheme (e.g., as described in § A). We define three atomic actions: (1) *request*(m, t), which corresponds to thread t requesting a mutex m ; (2) *acquire*(m, t), which corresponds to thread t acquiring mutex m ; and (3) *release*(m, t), which corresponds to thread t releasing a mutex m .

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 ; this is the k^{th} mutex acquisition made by t .

Expressing mutex acquisitions as triples emphasizes that they 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) ; k is retrieved by applying a function *index* to the pair (e.g., $k = \text{index}(m, t)$).

Two mutex acquisitions are defined to be *conflicting* if they are made by different threads on the same mutex. In general, the order in which conflicting mutex acquisitions are made may affect the result of a computation.

Definition 2 (History) The history H^r of a replica r is the sequence of mutex acquisitions by its threads at a given time. The notation $(m_i, t_i) \stackrel{H^r}{<} (m_j, t_j)$ indicates that (m_i, t_i) temporally precedes (m_j, t_j) in H^r .

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

Definition 3 (Causal Precedence) The causal precedence between two mutex acquisitions (m_i, t_i) and (m_j, t_j) in a history H , i.e., $(m_i, t_i) \stackrel{H}{\rightsquigarrow} (m_j, t_j)$, is defined as the transitive closure of the following relation:

1. $t_i = t_j \wedge (m_i, t_i) \stackrel{H}{<} (m_j, t_j)$, for mutexes acquired by the same thread, or
2. $m_i = m_j \wedge (m_i, t_i) \stackrel{H}{<} (m_j, t_j)$, for conflicting mutex acquisitions.

Causal precedence implies temporal precedence, while the reverse 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 [13]. Because concurrent events in distributed systems are not causally related, concurrent mutex acquisitions in a multithreaded process are those acquisitions whose actual order of execution does not affect the result of the computation. To preserve concurrency, we allow replicas to schedule concurrent mutex acquisitions independently. Based on the notion of causal precedence, the next definition introduces the causal set of a mutex acquisition, which represents all mutex acquisitions upon which a given mutex acquisition is causally dependent.

Definition 4 (Causal Set) *Given a mutex acquisition (m, t) in a history H , the causal set of (m, t) is the set $\theta_H(m, t) = \{(m', t') \in H \mid (m', t') \xrightarrow{H} (m, t)\} \cup \{(m, t)\}$.*

A deterministic scheduling algorithm must assume that threads behave deterministically between two consecutive mutex acquisitions. This assumption is somewhat similar to the piecewise deterministic assumption made by proponents of message-logging checkpointing [14]. While determinism is traditionally expressed in terms of state, the causal set is used as an abstraction to represent a thread's view of the replica's state at the moment of a given mutex acquisition.³ In this context, we refine the piecewise deterministic assumption made for message-logging checkpointing as follows:

Definition 5 (Piecewise Thread Determinism) *A thread t in a replica r is piecewise deterministic iff given its last mutex acquisition (m, t) , the behavior of t is uniquely determined by $\theta_{H^r}(m, t)$ and the replica's initial state S_0^r . In particular, from the initial state (i.e., before its first mutex acquisition), t 's behavior is uniquely determined by S_0^r .*

Because of this definition, outputs emitted by t between a mutex acquisition (m, t) and its next mutex acquisition are a function only of $\theta_{H^r}(m, t)$ and S_0^r . Moreover, race conditions are precluded. Observe that a thread's behavior in general depends on the inputs the thread receives. Although the above definition does not explicitly mention replica inputs, these can be incorporated in the model by requiring that corresponding threads of different replicas are supplied the same sequence of inputs at the same logical time.⁴

We now define the correctness properties of a deterministic scheduling algorithm. First however we introduce two predicates: (1) t requests m , which holds if thread t has requested a mutex m but has not acquired it yet (according to the action definitions given earlier); and (2) t owns m , which holds if thread t has acquired mutex m and has not released it yet. Formally, these predicates are defined as follows:⁵ t requests $m \equiv \neg \text{acquire}(m, t) \mathcal{S} \text{request}(m, t)$; and t owns $m \equiv$

³For example, suppose that a thread t 's behavior after each of its mutex acquisition (m, t, k) can be described as a function $\mathcal{F}((m, t, k), LS_t, SS_m)$, where LS_t and SS_m are, respectively, t 's local state and the replica shared state associated with m , both at the moment of (m, t, k) . Then, it can be shown that t 's behavior can be expressed as a function $\mathcal{G}(\theta_H(m, t, k), S_0^r)$, where S_0^r is the replica initial state.

⁴This is relatively simple for blocking I/O operations.

⁵The linear temporal logic symbol \mathcal{S} denotes *since*. $p \mathcal{S} q$ indicates that q was true in the past and p has been true from that moment until now.

$\neg \text{release}(m, t) \mathcal{S} \text{acquire}(m, t)$. For a given application, we also define a *mutex dependency graph* as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where:

1. $\mathcal{V} = \mathcal{T}$ is the set of vertices, $\mathcal{E} \subseteq \mathcal{T} \times \mathcal{T}$ is the set of edges, and
2. $(t, t') \in \mathcal{E}$ iff thread t requests a mutex owned by thread t' , i.e., $\exists m \in \mathcal{M}. (t \text{ requests } m \wedge t' \text{ owns } m)$.

The presence of a cycle in the mutex dependency graph is indicated by the following predicate *cycle*:

$$\text{cycle} \equiv \exists t_0 \dots t_{n-1} \in \mathcal{T} \exists m_0 \dots m_{n-1} \in \mathcal{M}. \\ \bigwedge_{i=0}^{n-1} (t_i \text{ requests } m_i \wedge t_{(i+1) \bmod n} \text{ owns } m_i).$$

Using the introduced predicates, the properties expected from a correct multithreaded application are formalized:

Definition 6 (Correct Application) *A multithreaded application with piecewise deterministic threads is defined to be correct iff:*

1. Each application thread releases only mutexes it owns;
2. If an application thread executes infinitely often,⁶ then the thread (a) eventually releases each mutex it acquires and (b) requests mutexes infinitely often;
3. The mutex dependency graph is acyclic.

The correctness of a deterministic scheduling algorithm is defined with respect a *correct* application as the conjunction of an Internal Correctness property, which defines the behavior of the algorithm only with respect to one replica, and an External Correctness property, which defines the behavior of the algorithm with respect to other replicas. These two properties are formally defined below.

Property 1 (Internal Correctness) *Given a replica r executing a correct application, the following conditions must always hold:*

1. (Mutual Exclusion) *At most one thread holds a given mutex: $\text{acquire}(m, t) < \text{acquire}(m, t') \rightarrow \text{acquire}(m, t) < \text{release}(m, t) < \text{acquire}(m, t')$;*
2. (No Lockout) *If a thread requests a mutex, then the thread will eventually⁷ acquire the mutex: $\text{request}(m, t) \rightarrow \diamond \text{acquire}(m, t)$.*

Property 2 (External Correctness) *Given two replicas r_1 and r_2 executing a correct application and started from the same initial state, two conditions must always hold:*

1. (Safety) *The causal sets of the mutexes acquired by both replicas are the same: $(m, t) \in H^{r_1} \cap H^{r_2} \rightarrow \theta_{H^{r_1}}(m, t) = \theta_{H^{r_2}}(m, t)$;*
2. (Liveness) *Any mutex acquisition made by r_1 is eventually made by r_2 : $(m, t) \in H^{r_1} \rightarrow \diamond (m, t) \in H^{r_2}$.*

In a replicated system, a deterministic scheduling algorithm is required to satisfy the above correctness properties for any pair of replicas.

⁶Given a predicate p , " p holds infinitely often" indicates that from now on p holds an infinite number of times (this is denoted with $\square \diamond p$ using linear temporal logic operators). In practice, parts (a) and (b) of the definition state how threads must behave if they are always making progress.

⁷The linear temporal logic symbol \diamond denotes *eventually*.

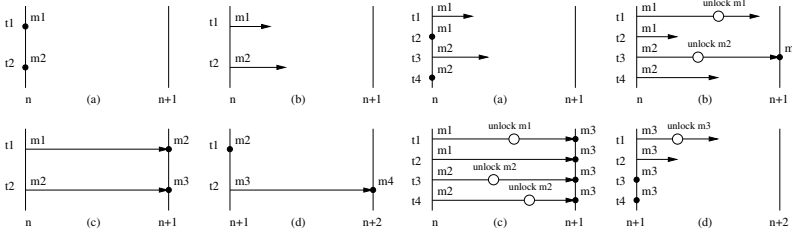


Figure 1. PDS-1 sample execution 1.

Figure 2. PDS-1 sample execution 2.

4. Specification of the PDS Algorithm

In the following, it is assumed that a total order relation ' $<$ ' is imposed on the set of replica threads \mathcal{T} , using which threads can be sorted; this order is the same for all replicas. Also, in the application threads the system calls `lock` and `unlock`, to acquire and release a mutex, are replaced with the functions `pds_lock` and `pds_unlock`.

For ease of understanding, we first present an overview of the PDS-1 algorithm, in which each thread can acquire at most one mutex per round. We then provide detailed specification of the PDS-2 algorithm, which improves concurrency by allowing a thread to acquire up to two mutexes per round.⁸ This section assumes that the set of application threads \mathcal{T} and the set of application mutexes \mathcal{M} are finite and do not change; this restriction is removed in § 4.3.

The PDS algorithm, as described in this paper, does not support *recursive* mutexes, i.e., mutexes that can be acquired consecutively times by their owner without being released. The algorithm can however be easily extended to handle such mutexes; furthermore, higher-level synchronization primitives (e.g. condition variables) can be implemented easily based on the mutex primitives presented in this paper.

4.1. PDS-1 Algorithm Overview

A replica's execution is broken into a sequence of rounds, and in a round each thread can acquire at most one mutex. On requesting a mutex, a thread t checks whether all other threads are suspended. If so, t triggers a new round; otherwise, t is suspended. When a new round is started, all threads' mutex requests are known, and therefore, a deterministic scheduling of mutex acquisitions naturally occurs: threads simultaneously requesting the same mutex acquire it according to increasing thread ids. Because all threads must have requested a mutex in order for the next round to fire, it is important that no thread have unbounded computation or blocking time between a mutex acquisition and the following mutex request. This is required by the definition of correct application (see § 3) and is further discussed in § 4.4.

Figure 1 shows an execution in which only two threads are considered. At beginning of round n , thread t_1 and t_2 have requested mutex m_1 and m_2 , respectively (indicated by small black circles in Figure 1(a)). Since the mutexes are different and no thread owns these mutexes, t_1 and t_2 can acquire them and run concurrently throughout round n (indicated by right-hand arrows in Figure 1(b)) until they both request the next mutex, namely

⁸Allowing a thread to acquire more than two mutexes per round leads to race conditions. Consequently, additional support must be provided. This is the subject of further study.

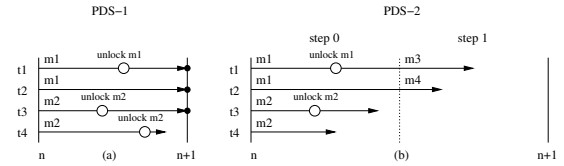


Figure 3. PDS-2 execution.

m_2 for t_1 and m_3 for t_2 (see Figure 1(c)). At this point, round $n + 1$ fires, after which thread t_2 can acquire m_3 but t_1 cannot acquire m_2 because this mutex is held by t_2 . Eventually, thread t_2 requests m_4 (see Figure 1(d)) and fires round $n + 2$. Thus t_2 is granted m_4 and, by executing, has a chance to release m_2 and so let t_1 execute as well.

Figure 2 presents a more complex execution scenario of the PDS-1 algorithm. At beginning of round n , threads t_1 and t_2 have requested m_1 , while t_3 and t_4 have requested m_2 . Threads t_1 and t_3 can execute concurrently because they have requested different mutexes and have the lowest ids with respect to the mutexes they request. Threads t_2 and t_4 remain suspended (see Figure 2(a)).

Later, t_1 releases m_1 and t_3 releases m_2 (indicated by white circles in Figure 2(b)); as a consequence, t_2 and t_4 can resume their execution. Thus, all threads can run concurrently. In Figure 2(b), thread t_3 has been suspended upon requesting m_3 . To ensure deterministic behavior, t_3 needs to wait for the other threads to “declare their intention” in terms of which mutex they want to acquire next.

In Figure 2(c) all threads have requested m_3 as the next mutex. At this point, round $n + 1$ fires and t_1 acquires m_3 . In Figure 2(d), when t_1 releases m_3 , t_2 is granted m_3 and so runs concurrently with t_1 .

4.2. PDS-2 Algorithm

In this section, the PDS-1 algorithm is extended (to improve concurrency) by allowing each thread to acquire up to two mutexes per round. Figure 3(a) shows an instant in the PDS-1 execution represented in Figure 2 where, during round n , threads t_1, t_2 , and t_3 are suspended because the new round cannot fire unless t_4 , still running, requests its next mutex. This waiting is not necessary. Indeed, whichever mutex t_4 requests next, say m' , at the next round t_4 will be scheduled only after any thread with a lower id and requesting m' is scheduled and releases m' . Therefore, it is possible to determine a new mutex acquisition scheduling for t_1, t_2 , and t_3 before t_4 reaches the end of round n (as t_1, t_2 , and t_3 have ids lower than t_4). This potentially permits all threads to run again concurrently and reduces thread waiting time at the round boundary.

In the PDS-2 algorithm, each round is divided into two steps, and each thread can acquire at most two mutexes per round. Figure 3(b) shows the PDS-2 execution corresponding to the PDS-1 execution of Figure 3(a), where t_1 and t_2 are allowed to acquire their next mutex and proceed to step 1 before the higher id threads t_3 and t_4 have requested their first mutex for that round, i.e., have completed step 0.

Due to space limitations, an extensive description of the pseudocode (Figure 4) and a correctness proof for the PDS-2 algorithm are relegated to [15]. Besides deriving a paper-and-pencil formal proof (based on modelling the algorithm as an I/O automaton), we used the Spin model checker [16] to automatically verify the Internal Correctness Property of our pseudocode. We

are currently investigating how to automatically verify the External Correctness Property as well. This task is complicated by the difficulty of representing the causal set notion in a model checker, where the modeled system has to have finite state.

4.3. Dynamic Mutex and Thread Creation

To enable dynamic mutex creation, the changes to the PDS algorithm (pseudocode in Figure 4) are straightforward. It is sufficient to replace the set of all mutexes \mathcal{M} with a set of current mutexes $mutexes \subset \mathcal{M}$ and to introduce two functions, `pds_mutex_create` and `pds_mutex_destroy`, for adding and removing mutexes, respectively. No additional mechanism is required for maintaining strong replica consistency, since under the piecewise thread determinism assumption, threads create mutexes at the same logical time (i.e., at the same step of the same round) at different replicas.

Similarly, to enable dynamic thread creation, we introduce a set of the current threads $threads \subset \mathcal{T}$, which replaces \mathcal{T} in the pseudocode, and two additional functions, `pds_thr_create` and `pds_thr_exit`, for adding and removing threads, respectively. Unlike dynamic mutex creation, dynamic thread creation involves additional complexity for enforcing strong replica consistency. This is because of the arbitrary temporal separation between the time at which a parent thread t_p creates a child thread t_c (an event synchronous with the PDS algorithm) and the time of the actual execution of t_c 's first instruction (an event depending on thread scheduling performed by the operating system and, hence, asynchronous with the PDS algorithm).

To eliminate this asynchrony, t_c 's execution is resynchronized with PDS (at each replica) at the beginning of round $n+1$, where n is the round at which t_c is created by t_p . t_c 's first instruction waits for round $n+1$ to fire, and round $n+1$ can fire only when all threads (including t_c) are suspended. A similar synchronization is used for thread termination, so that threads are terminated at the beginning of the same rounds at all replicas. Moreover, when a thread t invokes `pds_thr_exit` before terminating, t is removed from $threads$; thus, at this time it is necessary to check whether a new round-firing is enabled.

As a final requirement, both `pds_thr_create` and `pds_mutex_create` must return the same thread/mutex ids for corresponding threads/mutexes of different replicas; this can be enforced by using a hierarchical thread/mutex naming scheme (see § A).

4.4. Ensuring Timely Mutex Requests

The PDS algorithm requires that each application thread must acquire mutexes infinitely often (see § 3) in order to guarantee that `new_round` actions fire infinitely often and, hence, application threads can always make progress. In addition, timely mutex requests are important for the algorithm's performance. The key factor in achieving high performance is the relative difference in the amount of time T that each individual thread spends between a mutex acquisition and the following mutex request (within the same thread). If all threads take the same time T , then the PDS algorithm provides high performance independently of the specific value of T . Noting that T is determined by the computation and the I/O that a thread performs, we briefly discuss cases in which an application may not provide timely mutex requests.

(1) If threads exhibit long periods of computation between two mutex requests, application threads can be instrumented by

```

TYPE  $\mathcal{M}, \mathcal{T}$ 
STATE
  mutex :  $\mathcal{M} \rightarrow \text{tuple of owner} : \mathcal{T} \cup \{\perp\}$ ;
  q : array[0..2] of seq( $\mathcal{T}$ )
  step :  $\mathcal{T} \rightarrow 0..2$ 
  decl : array[1..2] of  $2^{\mathcal{T}}$ 
  n_suspended :  $\mathbb{N}$ 
  gm :  $\mathcal{M} /* Private mutex used to serialize accesses to PDS code. */$ 
INITIALLY
   $\forall m \in \mathcal{M} \forall j \in 0..2. mutex(m).owner = \perp$ 
   $\wedge mutex(m).q(j) = \langle \rangle$ 
   $\forall t \in \mathcal{T}. step(t) = 0$ 
   $\forall j \in 1..2. decl(j) = \emptyset$ 
  n_suspended = 0
DEFINITIONS
  can_acquire( $m : \mathcal{M}, t : \mathcal{T}$ )  $\equiv$ 
    mutex(m).owner =  $\perp \wedge step(t) < 2 \wedge$ 
     $t = head(mutex(m).q(step(t))) \wedge$ 
    mutex(m).q(step(t))  $\neq \langle \rangle \wedge step(t) =$ 
     $min\{k \in 0..1 \mid mutex(m).q(k) \neq \langle \rangle\} \wedge$ 
     $(step(t) > 0 \rightarrow (\forall t' \in \mathcal{T} \mid t' < t. t' \in decl(step(t))))$ 
  next_thr( $m : \mathcal{M}$ )  $\equiv$ 
     $\begin{cases} \perp & \text{if } \forall t \in \mathcal{T}. \neg can\_acquire(m, t) \\ t & \text{if } can\_acquire(m, t) \end{cases}$ 
SYSTEM FUNCTIONS
  curr_thr( $t : \mathcal{T}$ ) Returns the current thread.
  lock( $m : \mathcal{M}$ ) Locks a mutex  $m$ .
  unlock( $m : \mathcal{M}$ ) Unlocks a mutex  $m$ .
  suspend( $m : \mathcal{M}$ ) Atomically releases a mutex  $m$  and suspends the current thread, which holds  $m$  when resumed.
  resume( $t : \mathcal{T}$ ) Resumes a thread  $t$ .
1: Procedure new_round( $t : \mathcal{T}$ )
2: for all  $m \in \mathcal{M}$  do
3:   mutex(m).q(0) := mutex(m).q(0)  $\hat{\cap}$ 
4:   mutex(m).q(1)  $\hat{\cap}$  mutex(m).q(2)
5:   mutex(m).q(1) :=  $\langle \rangle$ 
6:   mutex(m).q(2) :=  $\langle \rangle$ 
7: end for
8: for all  $t \in \mathcal{T}$  do
9:   step(t) := 0
10: end for
11: decl(1) :=  $\emptyset$ 
12: decl(2) :=  $\emptyset$ 
13: resume_thrs(t)
1: Procedure resume_thrs( $t : \mathcal{T}$ )
2: local nt :  $\mathcal{T}$ 
3: for all  $m \in \mathcal{M}$  do
4:   nt := next_thr(m)
5:   if  $nt \neq \perp \wedge nt \neq t$  then
6:     mutex(m).q(step(nt)) :=
7:     tail(mutex(m).q(step(nt)))
8:     n_suspended := n_suspended - 1
9:     mutex(m).owner := nt
10:    resume(nt)
11:   end if
12: end for
1: Procedure pds_lock( $m : \mathcal{M}$ )
2: local t :  $\mathcal{T}$ 
3: lock(gm)
4: t := curr_thr()
5: step(t) := step(t) + 1
6: decl(step(t)) := decl(step(t))  $\cup \{t\}$ 
7: mutex(m).q(step(t)) :=
8: sorted_insert(mutex(m).q(step(t)), t)
9: n_suspended := n_suspended + 1
10: resume_thrs(t)
11: if  $\neg can\_acquire(m, t) \wedge$ 
12:   n_suspended = # $\mathcal{T}$  then
13:   new_round(t)
14: end if
15: if  $\neg can\_acquire(m, t)$  then
16:   suspend(gm)
17: else
18:   mutex(m).q(step(t)) :=
19:   tail(mutex(m).q(step(t)))
20:   n_suspended := n_suspended - 1
21:   mutex(m).owner := t
22: end if
23: unlock(gm)
1: Procedure pds_unlock( $m : \mathcal{M}$ )
2: local nt :  $\mathcal{T}$ 
3: lock(gm)
4: mutex(m).owner :=  $\perp$ 
5: nt := next_thr(m)
6: if  $nt \neq \perp$  then
7:   mutex(m).q(step(nt)) :=
8:   (mutex(m).q(step(nt)))
9:   n_suspended := n_suspended + 1
10:   mutex(m).owner := nt
11:   resume(nt)
12: end if
13: unlock(gm)

```

Figure 4. PDS-2 algorithm.

inserting acquisitions on an artificial mutex so that these periods are broken into execution chunks of similar duration.

(2) If threads are suspended for data/connections to arrive, then in principle T can be arbitrary large. However, there is a range of applications in which a replicated server will be subjected to regular loads (e.g., consider a call-processing application where threads establish line connections for incoming calls). For these applications, new data/connections are regularly available to all threads and the PDS algorithm performs well. Solutions to handle late data/connection arrivals are discussed in [15].

4.5. Using the PDS Algorithm with Majority Voting

Because the PDS algorithm does not require inter-replica communication, no additional failure modes are introduced by using the algorithm beyond what one would have in a typical replicated system. For example, in an active replication scheme with $2f + 1$ replicas, up to f arbitrary failures can be masked by majority voting on replica outputs.

The issue of majority voting for single-threaded replicas is relatively straightforward. Here, majority voting requires replica *output consistency*, for which two conditions must be met: (1) *input consistency*, in which the input requests are identical and

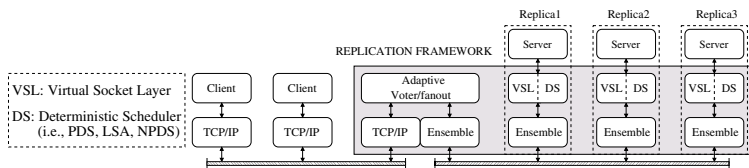


Figure 5. Experimental setup.

delivered to correct replicas in the same order [17], 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 [18].

For multithreaded replicas, the condition of replica determinism is replaced with the condition of *piecewise thread determinism* (discussed in § 3). In addition, output consistency needs to hold only with respect to corresponding threads across replicas, while the voter must compare replica outputs on a per-thread basis. Output comparison on a per-thread basis is necessary because the PDS algorithm (similarly to the LSA algorithm [6]) synchronizes replicas on shared state updates. As a consequence, it does not guarantee any ordering on the outputs produced by different threads at different replicas. Threads can be scheduled in a different order and/or executed with different timing. To support output comparison on a per-thread basis it is necessary that replicas (1) use identical thread ids for corresponding threads and (2) tag their outputs with the logical id of the thread generating the output.

5. Performance-Dependability Trade-offs

Thus far we have provided the details of the PDS algorithm, formally specified it, and (in [15]) verified its correctness. The next sections discuss an experimental evaluation of the algorithm from both performance and dependability perspectives. The necessity for performance assessment is clear. Although a level of dependability assessment has been achieved formally, the process is not complete unless the algorithm is also evaluated experimentally, especially with regard to its response under a wide range of failures. The goal of the dependability assessment is to evaluate the fault resilience of the algorithm. Whereas the goal of a deterministic scheduling algorithm is to support failure masking, it is critical that the algorithm itself does not constitute a major source of failures in the system.

The analysis of the PDS algorithm's performance and dependability characteristics is pursued through an experimental study of the performance-dependability trade-offs involved in selecting deterministic scheduling algorithms when replicating multithreaded applications. The considered algorithms include the PDS algorithm (introduced in this paper), the LSA algorithm (introduced in [6]), and a nonpreemptive deterministic scheduling (NPDS) algorithm (based on the Transactional Drago's algorithm [5] proposed in the context of transactional applications).

5.1. Performance Evaluation

In this section, performance is evaluated running a synthetic benchmark (described below), which emulates different levels of parallelism in a multithreaded replica execution in an active replication configuration with majority voting. Two performance measures are used: (1) the replicated server's throughput (number of client requests served per second) and (2) the repli-

Table 1. Services' Description.

Service Type	Sequence of Activities (and Corresponding Scenario)
A	lock m_0 —lock m_1 —unlock m_0 —unlock m_1 —I/O—lock m_2 —unlock m_2 —I/O (combination of accesses to a two- and one-level data structure).
B	lock m_3 —lock m_4 —I/O—unlock m_3 —unlock m_4 —I/O (access to a two-level data structure).
C	lock m_5 —unlock m_5 —I/O—lock m_5 —unlock m_5 —I/O (two consecutive accesses to the same one-level data structure).
D	lock m_6 —unlock m_6 —I/O—lock m_7 —unlock m_7 —I/O (two consecutive accesses to two different one-level data structures).

cated server's latency (time interval between sending a request and receiving a response to this request), as seen by a client.

The experimental setup (see Figure 5) consists of two Ethernet 100 Mbps LANs, one connecting the clients to a voter/fanout process and the other connecting the voter/fanout process to three replicas. Replicas and voter execute on Pentium III 500 MHz-based machines running Linux 2.4, and Ensemble 1.38 [8] is used for group communication. The replication framework employed is the Virtual Socket Layer [15], which provides transparent active replication to socket-based applications.

Synthetic Benchmark. A synthetic benchmark models a multithreaded, networked server in which 10 worker threads serve requests coming concurrently from 15 clients. By setting the number of incoming client requests greater than the number of server threads, we can study the maximum server throughput. A client request is composed of a header, which specifies the (random) type of the service requested (described later), and a payload message, which has random contents and size (the size is uniformly distributed between 0 and 1000 bytes). Each client continuously generates a random request and waits for the response to arrive.

Serving a client request involves two steps: (1) fast conversion of the payload (lower-to-upper case string conversion is used in the experiments), an operation used solely to enable the voter to verify that corresponding server threads at different replicas serve the same client requests (any inconsistency is detected as a value fault, since the conversion result is included in the server response), and (2) execution of a sequence of (i) mutex acquisitions modeling accesses to shared data and (ii) I/O activities (emulated by thread suspension) modeling server access to a persistent storage, e.g., data base. The two activities are interleaved to model variable workloads and allow different parallelisms in thread execution. The duration of I/O activity is a random variable uniformly distributed in $[0, D_{max}]$, where D_{max} is a parameter that can be varied to emulate different levels of parallelism offered by the benchmark: the larger D_{max} , the more parallelism in thread execution.

Using the above workload model, four types of services are generated (see Table 1). For example, to serve a request of type A, a server thread t first acquires mutex m_0 and mutex m_1 ; then, the mutexes are released in the same order. This acquisition sequence is typical of accesses to two-level data structures, e.g., a hash table, where m_0 protects the table of pointers to the collision lists, and m_1 protects the particular collision list containing the requested element. Next, I/O is performed, after which thread t acquires m_2 and releases it; this is to model an access to a different data structure. Finally, a second I/O is performed.

By having each client generate a random mix of these service types, we model a realistic scenario in which some server threads will serve the same request type and so will contend for the same mutexes, while other threads execute independently of each other.

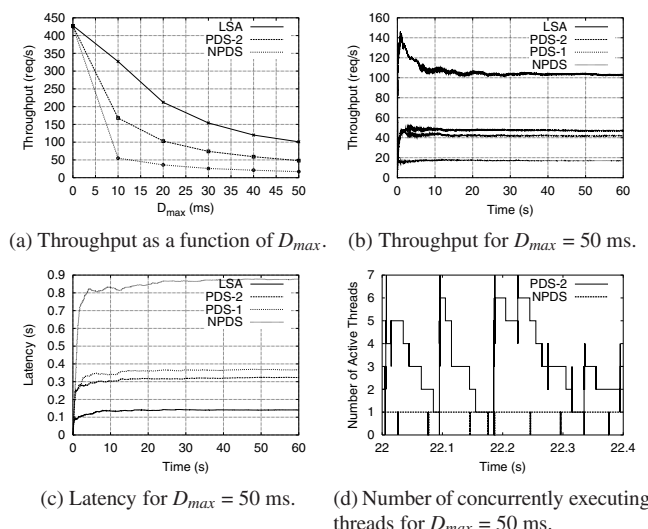


Figure 6. Triplicated server.

Triplicated Server Experiments. Figure 6(a) shows the triplicated server’s throughput, measured at the client side, as a function of D_{max} . All algorithms perform almost identically when D_{max} is zero: since replicas run on single-processor machines, threads are serialized. As D_{max} increases, i.e., the potential concurrency in thread execution increases, the LSA algorithm performs best, while the NPDS algorithm performs poorly (LSA provides about five times more throughput than NPDS). The PDS algorithm stays in the middle between the LSA and the NPDS (PDS provides about two times more throughput than NPDS); therefore, PDS can be considered a compromise between aggressive concurrency at the expense of inter-replica communication (LSA), which gives better performance, and independent replica execution at expense of concurrency (NPDS), which provides better error resilience (as discussed in § 5.2). The throughput of all algorithms decreases as D_{max} grows because the service time increases while the number of server threads is fixed.

Figures 6(b) and 6(c) show, respectively, server throughput and latency as functions of experiment execution time for D_{max} set to 50 ms. The two graphs, in addition to reinforcing the conclusions (from Figure 6(a)) on the relative performance of the tested algorithms, show also that PDS-2 provides about 12% more throughput than PDS-1 (this holds also for lower values of D_{max}). To explain the large diversity in terms of performance between the NPDS and PDS algorithms, Figure 6(d) shows the number of concurrently executing threads (i.e., those threads not suspended by the scheduling algorithm) for the PDS-2 algorithm and the NPDS algorithm. Observe that while the PDS-2 curve has a periodic tooth-saw shape (which is in synchrony with the algorithm round firing), the NPDS curve is almost always one, occasionally decreasing to zero when the running thread invokes the function `accept` for accepting a new client connection and another thread is scheduled.

The above experiments show the LSA algorithm performing better than the PDS algorithm. It should be noted, though, that this result can be inverted if inter-replica communication is costly, e.g., when the network bandwidth is scarce or the number of replicas is large. On the other hand, the NPDS algorithm always performs worse than both LSA and PDS, as long as there is parallelism in replica threads’ execution that can be exploited.

5.2. Dependability Evaluation

This section provides a dependability assessment and comparison of the PDS and the LSA algorithms using software-based error injection. Three dependability measures are used: (1) the number of catastrophic failures (cases in which the entire replicated system fails⁹), (2) the replica’s error-manifestation ratio (ratio between manifested and injected errors), and (3) the replica’s manifested-to-activated¹⁰ error ratio (when available). The number of catastrophic failures must be minimized in highly available systems. The error manifestation ratio characterizes the likelihood that an error causes a failure (including a single replica and an entire system failure). It can be used to calculate, given the error arrival rate, the replicated system availability. The manifested-to-activated ratio provides a closer look into the error sensitivity of a replica code. We now summarize the major findings from our error-injection experiments:

1. About 36000 single-bit errors were injected uniformly into text, data, and heap¹¹ segments of a replica process (including Ensemble [8], which provides group communication primitives). The results indicate a smaller error sensitivity for a PDS replica than for an LSA replica. The difference is due to the inter-replica communication required by LSA.
2. About 2800 single-bit errors were injected uniformly in the portion of a replica’s text segment corresponding to a specific Ensemble function that is used by both a PDS and an LSA replica. We observed 81 catastrophic failures, which shows that Ensemble’s fail silence violations constitute an issue for highly available systems.
3. In (2), we did not observe statistical difference, with respect to catastrophic failures, between LSA and PDS. Therefore, in an LSA leader, we injected errors into Ensemble functions solely used in LSA (note, all Ensemble functions used by PDS are also used by an LSA replica). Thirty-one catastrophic failures were observed in 3600 injections, which confirms that an LSA replica is more sensitive to catastrophic failures.
4. Importantly, errors originating in the reliable communication layer do propagate and lead to catastrophic failures of the entire replicated system. This is reported not as an indictment of Ensemble (which is a well-engineered product), but to point out that these failures are due to fail silence violations and error propagations, which are out of the scope of the usual assumption of crash/omission failures. In addition, simply using protocols capable of handling application value errors (e.g., interactive consistency) will not help cope with errors originating in the communication layer. (The catastrophic failures we observed are due to a corrupted Ensemble header in the messages exchanged.) To limit or prevent error propagations across the network, the middleware on which fault-tolerant techniques are based (in our case, the reliable communication layer), must itself be fault-tolerant [19].

5.2.1. Injections into a Replica Process

A first set of error injection experiments was conducted to assess the impact of errors in a replica’s text, data, and heap memory

⁹A replicated system fails if the voter fails or a majority of replicas fail.

¹⁰An error is activated if the injected data/instruction is used/executed.

¹¹More than one error may be injected during a single heap experiment.

Table 2. Error models.

Error Model	Description
TEXT	A single bit in the text segment of the target replica is flipped.
DATA	A single bit in the data segment of the target replica is flipped.
HEAP	A bit in allocated regions of the heap memory of the target replica is flipped periodically, until the replica terminates or crashes. Note that more than one error may be injected during a single experiment.

Table 3. Outcome categories.

<i>Crash Failure</i>	<i>SIGNAL</i> , the operating system terminates the target replica by sending a signal (e.g., SIGSEGV, SIGILL, SIGBUS, SIGFPE). <i>ASSERT</i> , the target replica shuts itself down owing to an internal check violation. <i>HANG</i> , the target replica does not terminate but no output is produced to the voter.
<i>Fail Silence</i>	<i>VALUE ERROR</i> , the target replica produces a value different from other, non-faulty replicas. <i>TWO HUNG FOLLOWERS</i> , the target replica (LSA leader) sends output to the voter but stops sending mutex tables to followers. [†]
<i>Violation</i>	<i>LEADER IMPERSONATION</i> , as the previous case except that no output is sent to the voter (the leader “impersonates” a follower). [‡] <i>CATASTROPHIC FAILURE</i> , the entire replicated system fails.

[†] A mutex table is a fragment of the leader’s replica history H^l and is continuously sent to followers so that the followers can enforce the same causal relations as the leader on their threads’ mutex acquisitions. The LSA voter detects this case by noting that a majority of replicas are hung.

[‡] The LSA voter detects this case by noting that no replica generates output before a dedicated timer expires, although output is expected.

segments. A replica here includes the application benchmark discussed in § 5.1, the PDS/LSA algorithm, the Virtual Socket Layer, and Ensemble. NFTAPE [20], a software framework for conducting automated fault/error injection experiments, was used to conduct the tests.

The error models considered are summarized in Table 2 and represent a combination of those used in several past experimental studies [21, 22]. By injecting single bits in the targeted replica, we emulate errors in the main memory, the cache, the processor execution buffer, and the processor execution core, as well as errors occurring during the transmission over a bus. Previous research on microprocessors [23] has shown that most (90–99%) device-level transients can be modeled as logic-level, single-bit errors. Data on operational errors also shows that a majority of errors in the field are single-bit errors.

Manifested errors are divided in two major outcome categories: (1) *crash failures*, in which the injected replica stops executing and no incorrect state transition is performed before the failure, and (2) *fail silence violations*, in which the injected replica performs incorrect state transitions.¹² The two categories and their corresponding subcategories are reported in Table 3.

Table 4 reports the results from error injection experiments for both PDS and LSA algorithms (for LSA, we distinguish between leader and follower injections) and for each error model listed in Table 2. During an experiment, each of the 15 clients sends 10 requests (generated as explained in § 5.1) and then terminates. This setup permits us to observe the system for a sufficient amount of time after an error is injected (about 30 seconds). The experiment concludes when either all clients terminate or a catastrophic failure occurs. The system is reset between two experiments.

With the exception of five cases, the system is able to recover from the injected failure. In the case of PDS, the voter masks the failure. In the case of LSA, if the leader fails, followers successfully elect a new leader after the failed leader is excluded from the system; if a follower fails, the voter masks the failure.

¹²This definition of fail silence violation is consistent with [24]. This failure type covers cases such as corrupted data saved on persistent storage or corrupted message sent to other nodes.

In discussing further the error-injection results, we distinguish between failures masked by the voter and catastrophic failures.

Failures Masked by the Voter. Text injections show a slightly larger error-manifestation ratio for LSA than for PDS: 10%, 8.3%, and 7.5% for an LSA leader, LSA follower, and PDS, respectively. The manifested-to-activated error ratios give a similar conclusion: 73%, 65%, 68% for an LSA leader, LSA follower, and PDS, respectively. This ratio variation could be explained by the different complexity of the two algorithms (with PDS being simpler than LSA). Because the difference in the algorithms’ code size (14K for PDS and 25K for LSA) is small compared to the total replica code size (900K) and the errors are injected uniformly, we argue that the major cause for variations in the observed error manifestations is the *different uses of Ensemble* (with code size of about 740K). While PDS and LSA replicas both use Ensemble to communicate with the voter, an LSA replica also uses Ensemble for passing the order of mutex acquisitions from the leader to the followers. Profiling the Ensemble usage shows that a PDS replica and an LSA replica invoke, respectively, 343 and 391 Ensemble functions.

Data injections show a very low error-manifestation ratio. This is because a large part of the data segment (405K in total, 390K of which is part of Ensemble) is not used during normal execution. As a result, errors in the data segment do not contribute noticeably to the number of failures in the system.

Heap injections show an error-manifestation ratio for LSA that is about twice that for PDS. The reason can be found in the more extensive use of dynamic memory by (1) the LSA leader, which stores the mutex acquisition order on the heap memory, (2) the LSA followers, which store the leader-decided order of mutex acquisitions in dynamic data structures (projection queues), and (3) Ensemble (for both leader and followers), which uses heap memory for internal message buffering and management support of the leader-to-follower communication. The observed larger error sensitivity of a follower is because a follower not only collects (in the projection queues) the leader-dictated order of mutex acquisitions, but actively applies it in scheduling threads’ executions. Therefore, corruption of the projection queues results in more crashes or divergent behavior of the follower, where divergent behavior manifests as a greater percentage of value errors.

Thus, the experiments show that an LSA-based replicated system is more sensitive to voter-masked failures than a PDS-based replicated system. Note that, although these failures do not cause errors to propagate to clients, they do impact a replicated system’s availability.

Catastrophic Failures. Although the above discussion indicates that the PDS thread-scheduling strategy has a higher error resilience than the LSA strategy, the most important difference between the two algorithms appears when analyzing catastrophic failures. Because these failures cannot be masked by the voter, it is crucial to prevent them in a replicated system. Two replicated systems can be judged by their ability to avoid this type of failures. In the experiments conducted, we observed five catastrophic failures occurring through the Ensemble communication layer, all of which were due to error propagation. They are described below.

PDS experiments. An error injected in the Ensemble’s message routing module (Unsigned) of the targeted replica caused

Table 4. Error injection results.

Error Model	Total Injected Errors	Total Activated Errors	Total [†] Manifested Errors	Manifested Errors [‡]							
				Crash Failures			Fail Silence Violations				
				SIGNAL	ASSERT	HANG	VAL ERR	2 H FOLL	LEAD IMP	CATAST FAIL	
PDS	TEXT	5224	583	394 (7.5%)	334 (85%)	21 (5.3%)	24 (6.1%)	14 (3.6%)	N/A	N/A	1 ^{††} (0.25%)
	DATA	2152	N/A	6 (0.28%)	5	1	0	0	N/A	N/A	0
	HEAP	9158	N/A	869 (9.5%)	782 (90%)	0	32 (3.7%)	55 (6.3%)	N/A	N/A	0
LSA-L	TEXT	5224	728	528 (10%)	447 (85%)	20 (3.8%)	16 (3.0%)	9 (1.7%)	27 (5.1%)	6 (1.1%)	3 [‡] (0.57%)
	DATA	2139	N/A	5 (0.23%)	4	1	0	0	0	0	0
	HEAP	2036	N/A	523 (26%)	501 (96%)	0	3 (0.57%)	8 (1.5%)	11 (2.1%)	0	0
LSA-F	TEXT	5144	659	429 (8.3%)	402 (94%)	12 (2.8%)	11 (2.6%)	3 (0.70%)	N/A	N/A	1 ^{†††} (0.23%)
	DATA	2153	N/A	5 (0.23%)	5	0	0	0	N/A	N/A	0
	HEAP	3010	N/A	961 (32%)	927 (96%)	0	9 (0.94%)	25(2.6%)	N/A	N/A	0

[†] The error-manifestation ratio (i.e., ratio between manifested and injected errors) is shown in parentheses.

[‡] The percentage of the particular manifestation type with respect to the total number of manifested errors is shown in parentheses.

the voter to crash in the Ensemble’s point-to-point communication module (`Pt2Pt`), but the injected replica did not crash.

LSA-leader experiments. Three catastrophic failures were caused by errors originating from an Ensemble function used by the LSA leader. (1) An error injected in the intra-group failures-and-view module (`Intra`) of the leader caused the voter to have an inconsistent group membership view with respect to other replicas,¹³ which violates the properties of reliable group communication. (2) An error injected in the connection management module (`Conn`) of the leader caused the leader to hang and the two followers to crash in their reliable, FIFO broadcast module (`MnAk`). (3) An error injected in the `Unsigned` module of the leader caused the two followers to crash in their `MnAk` module and the voter to crash in its `Pt2Pt` module.

LSA-follower experiments. An error injected in the Ensemble’s function `extern__rec` of the targeted follower caused the voter and the other two replicas to raise an exception due to a corrupted control flow packet header. This function handles the interaction between the the high-level part of Ensemble (e.g., reliable communication algorithms) written in ML and the low-level part (e.g., sockets) written in C.

5.2.2. Injections into Ensemble

To investigate further the sensitivity of the two algorithms to catastrophic failures, a new set of text error injections was performed targeting a specific function of Ensemble, `extern__rec` (1.4K code size). This function was selected because it was heavily used and generated a large number of catastrophic failures. Results from Table 5 reinforce our conclusions on the greater error sensitivity of LSA with respect to PDS: the error-manifestation ratio is 43% for LSA and 30% for PDS, while the manifested-to-activated error ratio is 87% for LSA and 82% for PDS. In a majority of catastrophic failures, the error originating from the injected replica caused other replicas and/or the voter to crash due to segmentation fault (25 cases for PDS and 26 cases for LSA). In the remaining cases (16 for PDS and 14 for LSA), other replicas and/or the voter terminated due to an Ensemble-generated exception (e.g., due to corrupted packet header). In a large number of catastrophic failures (22 for PDS and 30 for LSA), the injected replica did not crash.

In the previous experiments, we did not observe a statistical difference, with respect to catastrophic failures, between LSA and PDS. Therefore, a final set of experiments was conducted targeting, in an LSA leader replica, some of the Ensemble functions that, during normal execution (i.e., when no view change

¹³This inconsistency caused to the voter to receive a message from a replica that was not member of the group seen by the voter. On detecting this condition, the voter terminated by raising an exception.

occurs), are used by an LSA leader replica but not by a PDS replica. (Note that all Ensemble functions used by a PDS replica are also used by an LSA leader replica). These functions were selected from the Ensemble modules `Addr` (for network address management), `Conn`, and `Hot` (Ensemble’s C interface to user applications). The functions and their corresponding injection results¹⁴ are presented in Table 6. The results indicate that a significant number of catastrophic failures originate from these functions and, hence, that an LSA-based replicated system is more likely to exhibit catastrophic failures than a PDS-based replicated system. The voter did not fail in any of the catastrophic failures observed, which confirms that these failures are due to leader-to-follower communication.

5.3. Lesson Learned

Performance and failure analysis of the deterministic schedulers PDS, LSA, and NPDS shows the following:

1. LSA strategy provides the best performance (in terms of throughput and latency in response to client requests) at the expense of availability (measured in terms of resilience to errors). Because LSA relies on an inter-replica communication channel for efficient mutex acquisition scheduling, LSA is more sensitive to the underlying communication layer’s fail silence violations than is PDS. This leads to a larger number of catastrophic failures for LSA than for PDS. If minimizing downtime is crucial (as for highly available systems), PDS is a more appropriate choice than LSA. If performance concerns have priority over minimizing downtime, then LSA can be preferred to PDS.
2. NPDS strategy provides correct execution through serialization, which eliminates the benefit of multithreading and results in poor performance compared to PDS and LSA strategies. Although we did not explicitly evaluate NPDS dependability characteristics, we argue that they are similar to those of PDS, especially with regard to catastrophic failures. This is because neither of the two algorithms uses inter-replica communication.

6. Conclusions

Replication schemes by their nature impose a significant performance overhead. Measurements reported for several existing approaches to replication indicate performance overheads ranging from three to ten times that of nonreplicated systems [1–3, 25]. Until recently, only single-thread applications were replicated, since multithreading does not easily conform to the

¹⁴Errors were injected (one per experiment) in each bit of each byte in the portion of the text segment corresponding to the selected functions.

Table 5. Text injections into Ensemble function `extern_rec`.

	Total Injected Errors	Total Activated Errors	Total Manifested Errors	Manifested Errors						
				Crash Failures			Fail Silence Violations			
				SIGNAL	ASSERT	HANG	VAL ERR	2 H FOLL	LEAD IMP	CATAST FAIL
PDS	1419	526	433	273	37	82	0	N/A	N/A	41
LSA-L	1419	709	616	452	16	108	0	1	0	40

Table 6. Text injections into Ensemble functions used by LSA replicas but not PDS replicas.

Total Injected Errors	Total Activated Errors	Total Manifested Errors	Manifested Errors						
			Crash Failures			Fail Silence Violations			
			SIGNAL	ASSERT	HANG	VAL ERR	2 H FOLL	LEAD IMP	CATAST FAIL
3629	2667	2191	1589	356	200	15	0	0	31

state machine approach [18] widely used in software replication. However, if the replicas are multithreaded, then performance overhead due to replication can be lessened. A simplistic approach is pursued by nonpreemptive deterministic schedulers (i.e., in Eternal [4] and in Transactional Drago [5]), which although providing correct execution, do not exploit concurrency in multithreaded replicas. In contrast, the Loose Synchronization Algorithm (LSA) [6], proposed by the authors in an earlier paper, captures the natural concurrency in a leader replica and projects it on follower replicas through inter-replica communication. The Preemptive Deterministic Scheduler (PDS) algorithm, proposed in this paper, removes the need for inter-replica communication yet preserves a large degree of replica concurrency. The absence of inter-replica communication gives PDS dependability advantages over LSA.

Acknowledgments

This work is supported in part by NSF grants CCR 00-86096 ITR and CCR 99-02026.

A. Appendix

This section discusses how to enforce the assumption that the same thread/mutex ids are associated with corresponding threads/mutexes of different replicas (see § 3). If, on all replicas, corresponding threads/mutexes are created/initialized by the same thread and in the same order in the context of this thread, then a hierarchical thread/mutex naming scheme can be employed as follows.

The logical thread id t of a thread is recursively defined as the sequence $t' \wedge tcc(t')$, where t' is the logical id of t 's parent and $tcc(t')$ is the value of a thread-creation counter owned by t' at the time of t 's creation. This counter is incremented each time t' spawns a new child thread. By convention, the logical thread id of the application main thread is $\langle 0 \rangle$.

The logical mutex id m of a mutex is given by the pair $\langle t, mcc(t) \rangle$, where t is the logical id of the thread that creates the mutex, and $mcc(t)$ is the value of a mutex creation counter owned by t at the time of the mutex creation. This counter is incremented each time t creates a new mutex.

References

- [1] M. Cukier et al. AQUA: An adaptive architecture that provides dependable distributed objects. In *Proc. of Int'l Symp. on Reliable Distributed Systems*, pages 245–253, 1998.
- [2] S. Pleisch and A. Schiper. FATOMAS: A fault-tolerant mobile agent system based on the agent-dependent approach. In *Proc. of Int'l Conf. on Dependable Systems and Networks*, pages 215–224, 2001.
- [3] G. D. Parrington et al. The design and implementation of Arjuna. *Computing Systems*, 8(2):255–308, 1995.

- [4] 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.
- [5] R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. Deterministic scheduling for transactional multithreaded replicas. In *Proc. of Int'l Symp. on Reliable Distributed Systems*, 2000.
- [6] C. Basile et al. Loose synchronization of multithreaded replicas. In *Proc. of Int'l Symp. on Reliable Distributed Systems*, 2002.
- [7] A. Goldberg et al. Transparent recovery of Mach applications. In *Usenix Mach Workshop*, pages 169–183, 1990.
- [8] M. Hayden. *The Ensemble System*. PhD thesis, Dept. of Computer Science, Cornell University, USA, 1997.
- [9] A. Borg et al. Fault tolerance under UNIX. *ACM Trans. on Computer Systems*, 7(1):1–24, 1989.
- [10] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. on Computer Systems*, 14(1):80–107, 1996.
- [11] P. A. Barrett et al. The Delta-4 extra performance architecture (XPA). In *FTCS-20*, pages 481–488, 1990.
- [12] D. Powell et al. GUARDS: A generic upgradable architecture for real-time dependable systems. *IEEE Trans. on Parallel and Distributed Systems*, 10(6):580–599, 1999.
- [13] O. Babaoglu and K. Marzullo. *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.
- [14] E. Elnozahy, D. Johnson, and Y. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical report, Carnegie Mellon University, 1996.
- [15] C. Basile, Z. Kalbarczyk, and R. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. Technical report, University of Illinois at Urbana-Champaign, 2003. <http://www.uiuc.edu/~cbasile/papers>.
- [16] G. Holzmann. The SPIN model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
- [17] F. Cristian et al. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118(1):158–179, 1995.
- [18] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.
- [19] K. Whisnant et al. An experimental evaluation of the REE SIFT environment for spaceborne applications. In *Proc. of Int'l Conf. on Dependable Systems and Networks*, pages 585–595, 2002.
- [20] D. Stott et al. Dependability assessment in distributed systems with lightweight fault injectors in NFTAPE. In *Proc. of Int'l Computer Performance and Dependability Symposium*, 2000.
- [21] E. Fuchs. Validating the fail-silence assumption of the MARS architecture. In *Proc. of 6th Dependable Computing for Critical Applications Conference*, pages 225–247, 1998.
- [22] H. Madeira and J.G.Silva. Experimental evaluation of the fail-silent behavior in computers without error masking. In *Proc. of Int'l Symp. on Fault-Tolerant Computing*, pages 350–359, 1994.
- [23] M. Rimen, J. Ohlsson, and J. Torin. On microprocessor error behavior modeling. In *Proc. of Int'l Symp. on Fault-Tolerant Computing*, 1994.
- [24] F. V. Brasileiro et al. Implementing fail-silent nodes for distributed systems. *IEEE Trans. on Computers*, 45(11):1226–1238, 1996.
- [25] R. Guerraoui et al. System support for object groups. In *ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, 1998.