Multithreading Strategies for Replicated Objects*

Jörg Domaschka[1], Thomas Bestfleisch[1], Franz J. Hauck[1], Hans P. Reiser[2], and Rüdiger Kapitza[3]

[1] Department of Distributed Systems, Ulm University, Germany
{joerg.domaschka,thomas.bestfleisch,franz.hauck}@uni-ulm.de
[2] LaSIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal
hans@di.fc.ul.pt
[3] Dept. of Comp. Sciences 4, University of Erlangen-Nürnberg, Germany
rrkapitz@cs.fau.de

Abstract. Replicating objects usually requires deterministic behaviour for maintaining a consistent state. Multithreading is a critical source of non-determinism, completely unsupported in most fault-tolerant middleware systems. Recent publications have defined deterministic scheduling algorithms that operate at the middleware level and allow multithreading for replicated objects. This approach avoids deadlocks, improves performance, and makes the development better resemble that of non-replicated objects. This paper surveys those algorithms and analyses their differences. It also defines extensions to two efficient multithreading algorithms to support nested invocations and condition variables with time-bounded wait operations similar to the Java synchronisation model. In addition, we provide an experimental evaluation and performance comparison of the algorithms, indicating the areas in which each algorithm performs best. We conclude that replication middleware should implement reconfigurable multithreading strategies, as there is no optimal one-size-fits-all solution.

1 Motivation

Object replication is an important mechanism for implementing reliable distributed applications. Many current object middleware systems support replication. For example, FT-CORBA [1] and Jgroup [2] are architectures for replicating CORBA and Java RMI objects, respectively.

In many application domains of replication, such as file systems and data bases, the aim of replication is data-centric. In contrast, the replication of objects leads to different requirements, as it not only requires consideration of the state of the objects, but also of their activity. For example, an object method can actively interact with external services, and concurrently executing methods might use mechanisms such as semaphores, monitors, and condition variables for

* This work has been supported by the EC through FP6 Integrated Project IST2006-0033576 (XtreemOS) and project NoE IST-4-026764-NOE (ReSIST), and by the FCT, through the Multiannual Funding Programme.

V. Issarny and R. Schantz (Eds.): Middleware 2008, LNCS 5346, pp. 104–123, 2008.

coordination. A replication infrastructure should impose as few constraints as possible on the object implementations. This way, existing non-replicated implementations can be re-used directly for replication, and the developer can make use of the programming model he is used to, without paying a lot of attention to replication-induced restrictions.

Object replication strategies are typically classified as *active* or *passive*. In active replication, all replicas individually execute all requests, and the assumption of deterministic replica behaviour guarantees state consistency. In passive replication, only a single primary executes method invocations, and then transfers state updates to secondary replicas. At first sight, this strategy eliminates the need for determinism. However, sending state updates synchronously after each state modification is expensive. Often, the primary state is only periodically transferred to secondary replicas, and a message log is used to store client requests that the primary has executed since the last checkpoint. A secondary replica has to have the same deterministic behaviour if it wants to obtain a state identical to that of a failed primary by re-executing requests from such a log.

Multithreaded execution is a source of non-determinism, as multiple threads might execute at unknown relative speeds and might modify the object state in an unpredictable order. Non-replicated objects usually use mechanisms such as locks to coordinate concurrent state modifications. Nevertheless, different replicas of an object can grant locks in a different order, thus causing inconsistencies between replicas. Most object replication infrastructures avoid this problem by executing methods strictly sequentially.

In the past few years several authors have proposed solutions that to some extent enable deterministic multithreading in replicated objects [3,4,5,6]. The main motivation for multithreading is either to improve the efficiency of replicated objects, thus reducing the performance difference between replicated and non-replicated applications, to avoid inherent deadlock problems of single-threaded executions [7], or to provide a programming model that is as close as possible to the non-replicated case. This paper builds upon these previous works by enhancing some known algorithms and by providing a comparative analysis. The specific contributions of this paper are as follows:

- It presents a survey of all existing multithreading strategies for replicated objects known to the authors, clearly stating their differences in objectives, assumptions, and achieved properties.
- It augments two previously known algorithms with extensions that permit the use of these algorithms in a broader application spectrum.
- It provides an experimental evaluation and performance comparison of all algorithms, indicating the areas in which each algorithm performs best, and demonstrating that a middleware should provide configurability of the multithreading strategy, as there is no best algorithm for all situations.

This paper is structured as follows. The next section discusses the necessity of multithreading in replicated objects. Section 3 discusses and compares the existing algorithms. Section 4 defines extended variants of two algorithms, PDS and LSA. Section 5 presents an experimental evaluation based on several use cases. Finally, Section 6 concludes.

2 Background and Related Work

Many distributed object replication systems, such as OGS [8] and GroupPac [9], do not support multithreading in replicated objects. Method invocation requests from clients are executed in a strictly sequential order; a request is processed only after the preceding request has been completed. This approach avoids any non-determinism that can arise from thread scheduling, and it provides implicit synchronization of state modifications, as multiple threads cannot attempt to modify the state concurrently.

There are, however, several reasons that argue for the use of multithreading in replicated objects [7]. First of all, multithreading can enhance performance. On one hand, the computational power of multi-CPU hardware can be utilized better. On the other hand, multithreading allows the system to process additional method invocations whenever the system becomes idle because the current thread has to wait, e.g., during external invocations.

Second, a single-threaded execution excludes coordination via condition variables. For example, a thread might want to interact with an external service by first issuing an asynchronous external request, and then wait on a condition variable for the notification by a call-back of the external service.

Third, nested invocations can cause deadlocks in a single-threaded model, if a thread synchronously calls an external service, which in turn invokes a method at the originator.

Using multithreading in replicated objects requires that appropriate steps be taken to remove non-determinism. This means that in spite of concurrent execution of threads, the order of conflicting state manipulations must be made deterministic. The implementation of the replicated object must use some means to coordinate state modifications that happen concurrently in multiple threads. The use of explicit locks or monitors is the most popular model, but other mechanisms, such as non-blocking or wait-free synchronization, can also be found [10]. In this paper, we assume that multithreaded objects use lock synchronization; if these objects are replicated, the replication infrastructure must make sure that locks are granted in a consistent order on all replicas.

Some existing research projects use a modified Java virtual machine to implement deterministic replication. For example, *Napper et al.* (based on a modified Sun JDK 1.2) [11] and *Friedman and Kama* (based on a modified JikesRVM) [12] use this approach. Other systems ensure determinism at an even lower system level. For example, MARS [13] is strictly time-driven and periodic at the hardware level, which makes all functional and timing behaviour deterministic. The features of such a platform can be used for deterministic replication [14]. All these systems can execute multiple threads concurrently. They all require specifically designed hardware, operating systems, or Java virtual machines to achieve determinism.

In this paper we focus on means to enforce determinism of multithreaded replicated services purely at the middleware level, without requiring special low-level support in operating system or virtual machine. Several algorithms have been proposed in this area. On the basis of work by *Jiménez-Peris et al.* [15], *Zhao et al.* [3] propose a strategy to execute a new request during the idle time caused by a nested invocation of the main thread. Our ADETS-SAT algorithm [6] extends this approach with support for reentrant locks, condition variables, and time bounds on wait operations. These strategies allow the execution of a new thread only if the previous thread suspends. ADETS-MAT [7] is an improved version of ADETS-SAT that enables the concurrent execution of multiple threads. Basile's *Loose Synchronization Algorithm* (LSA) [4] supports true multithreading on the basis of a leader-follower model. Basile's *Preemptive Deterministic Scheduling* algorithm (PDS) [5] allows the concurrent execution of a fixed set of threads in periodic rounds, without requiring communication for consistency. We compare these algorithms in Section 3.2.

We use our FT*flex* replication infrastructure on top of the Aspectix middleware [16,17] for evaluating the multithreading strategies. FT*flex* supports deterministic multithreading on the basis of a plug-in interface for configurable ADETS (Aspectix DEterministic Thread Scheduler) modules.

3 Comparison of Algorithms

In this section, we define a set of criteria that allow a characterization of the various algorithms. These criteria include the *coordination model*, the *external interaction model*, the *deployment*, and the *multithreading model*. On this basis, we subsequently provide a systematic comparison of algorithms.

3.1 Criteria

Coordination Model. There are numerous different mechanisms for coordinating multiple concurrent threads. We restrict the discussion to *locks*, *monitors*, and *Java synchronization*, as these are most prevalent in existing systems.

Locks are a basic coordination mechanism that allows protecting the access to resources, using two operations `lock` and `unlock`. *Reentrant locks* allow one thread to acquire the same lock multiple times.

Monitors are a synchronization mechanism defined by Hoare that provides implicit locking around monitor procedures. Condition variables within the monitor allow threads to suspend and temporarily release the lock while waiting for a condition. Any thread that causes the condition to be true can signal the waiting thread, which in turn atomically regains the lock and resumes. By mapping a monitor procedure to a pair of `lock`/`unlock` operations, any scheduling algorithm for reentrant locks can be used for applications that use monitors for coordination. However, monitors in addition require support for condition variables by the scheduling algorithm.

Native Java synchronization uses a concept similar to Hoare monitors. It restricts the model by defining a single implicit condition variable for each monitor (instead of an arbitrary number). On the other hand, it supports time-bound wait operations, which allow a waiting thread to resume after a specific timeout.

Locks and monitors provide sufficient support to protect state modifications that should be made atomically. Condition variables are useful in situations in which threads wait for callbacks from external services and for coordinating in producer/consumer scenarios.

External Interaction Model. In terms of external interaction, we can distinguish between scheduling strategies that support *none (NO)*, and strategies that support *nested invocations (NI)* and *callbacks (CB)*. A nested invocation is an invocation of a function of a service B, issued by a service A during the execution of a client request. A callback is an invocation of a function of service A triggered by the nested invocation from A to a service B.

If a scheduling algorithm is deadlock-free for arbitrary nested invocations, it will also be deadlock-free for callbacks, which are a special kind of nested invocation. On the other hand, a scheduling algorithm can use thread IDs to detect that an incoming nested invocation belongs to an existing local (blocked) thread and thus identify it as a callback. Algorithms on this basis can supports deadlock-free callbacks, but are not necessarily deadlock-free for other nested invocations.

Deployment. Implementations of objects typically use either language-internal coordination mechanisms (such as in the case of Java) or invocations of synchronization methods of an external library (such as the pthread library for C++ objects). If such object implementations are deployed in a multithreading replication infrastructure, the synchronization methods have to be adjusted to interact with the replication infrastructure.

The simplest way is to provide *no support* at all, forcing developers to modify the implementation accordingly to interact with the replication infrastructure. A better alternative is an automated approach. Either, *code transformation* can be applied to alter the replica code automatically (as proposed for ADETS-SAT [6]). Or, *low-level interception* can redirect external library calls to the replication infrastructure (as used in Eternal [18]).

The deployment strategy could be considered as a concern orthogonal to deterministic scheduling. However, in ongoing work [19] we demonstrate that code analysis and transformation allows improving concurrency on the basis of prediction of future synchronization steps.

Multithreading Models. We classify the multithreading support of middleware infrastructures into four categories: *single thread (S)*, *single logical thread (SL)*, *single active thread (SA)*, and *multiple active threads (MA)*.

The *S* model is the simplest variant, in which the middleware starts executing a request R_{i+1} only after request R_i has fully completed its execution.

In the *SL* model, a single logical thread of execution exists. In a chain of nested invocations, the logical thread may call methods of the same object multiple times. For example, if a thread that executes m_{A1} at object A starts a chain of nested invocations that ultimately calls method m_{A2} at object A, the object A can detect that the invocation m_{A2} belongs to the same logical thread as m_{A1}. Thus, object A can execute m_{A2} using an additional physical thread. In a sequential execution model without the logical thread abstraction, nested invocations finally targeting the same object would cause a deadlock.

In the *SA* model, multiple independent physical threads can exist within a replica. Only one of them may be active at a time, while all other threads are blocked (e.g., waiting for a lock or for the return from a nested invocation). Consistency is obtained by a deterministic selection of the active thread. A running active thread is not preempted; if the active thread blocks or terminates, a deterministic strategy is required to resume one of the existing threads or to create a new active thread for handling the next request. If the strategy guarantees that the same choice is made in all replicas, consistency will be maintained.

The *SA* model does not require the identification of logical threads. However, such identification is mandatory if a system wants to support reentrant locks. For example, a method m_{A2}, called by m_{A1}, might want to acquire a lock already held by m_{A1}. We denote a *SA* model with appropriate logical thread identification as *SA+L*.

In *MA*, multiple threads may exist and run in parallel. Multiple threads may either be simultaneously active in a multi-CPU or multi-core CPU setting, or a low-level thread scheduler may execute them on a single CPU with preemption. To maintain consistency in true multithreading, all access to shared data structure needs to be made in a consistent order. The number of threads may be restricted by an algorithm.

3.2 Algorithms

This section discusses several different algorithms. Table 1 summarizes their different properties and models compared to a pure sequential execution.

SL in Eternal. The Eternal middleware was the first system to support the SL model [18] An infrastructure can support this model by tagging nested invocations with context information that identifies the originating logical thread.

SA and ADETS-SAT. Applying the approach of Jiménez-Peris et al. [15] to a CORBA middleware, Zhao et al. [3] implemented an algorithm for the SA

Table 1. Overview of multithreading algorithms and their properties

	Coordination	Deadl.-Free Interaction	Deployment	Multithreading
SEQ	implicit	NO	–	S
Eternal	implicit	CB	interception	SL
SAT	Locks	NI+CB	interception	SA
ADETS-SAT	Java	NI+CB	transformation	SA+L
ADETS-MAT	Java	NI+CB	transformation	MA
LSA	Locks/Monitor	NI+CB	manual	MA
PDS	Locks	NO	manual	MA (restr.)

model in Eternal. ADETS-SAT [6] is an extension of this algorithm that also offers support for reentrant locks and for Java condition variables. The algorithm ensures that threads calling `wait()` are enqueued in a deterministic manner and dequeued deterministically at a `notify()`.

ADETS-MAT. The ADETS-MAT algorithm [7] works similar to ADETS-SAT, but instead of using only a single active thread, it allows additional concurrency. Beside a primary thread that works similar to the active thread in ADETS-SAT, multiple secondary threads are executed concurrently as long as they do not request additional mutex locks. ADETS-MAT requires no communication for granting locks, threads can be created at any time by client requests, and no restrictions are made on the number and frequency in which a thread requests locks. Concurrency is constrained by the fact that only the primary thread can acquire locks. It fully supports the native synchronization model of the Java programming language. One of the main objectives of the algorithm is to use the idle time during nested invocations.

LSA. In LSA [4], a single replica works as primary node. This node can execute an arbitrary number of threads without restrictions, and records the order in which locks are granted to threads as a sequence of *(lock, thread)* pairs. It broadcasts this data structure to all other replicas periodically. All follower nodes suspend threads that request a lock until the corresponding broadcast is received. While the basic operation of LSA is very simple, it requires a strategy to handle the failure of the primary node. *Basile et al.* define such a fail-over algorithm for crash failures as well as for Byzantine failures. Failure handling requires additional communication between replicas to maintain consistent scheduling. This is a significant difference to other algorithms that do not need any additional computation or communication to handle node failures.

PDS. *Basile et al.* have defined two variants of the PDS algorithm, PDS-1 and PDS-2 [5]. Both algorithms work in sequential rounds. In PDS-1, each thread can acquire at most one mutex per round. A thread is suspended when it requests a mutex; as soon as all threads are suspended, a new round is started. As the mutex requests of all threads are known at the beginning of the round, the mutexes can be assigned deterministically to all threads. If multiple threads request the same lock, they get the lock in increasing thread ID order. For example, if two threads T_1 and T_2 have both requested a mutex m, T_1 may execute and T_2 remains suspended. As soon as T_1 unlocks m, T_2 may execute concurrently with T_1. If T_1 suspends in the current round without unlocking m, T_2 remains suspended.

The PDS-2 variant improves concurrency by allowing threads to acquire up to two locks per round. A round is divided into two phases. Initially, a round starts execution in phase-1 in the same way as PDS-1, granting mutexes according to requests made before the start of the round. If a thread requests a new mutex during phase-1, it is not immediately suspended (as it would be in PDS-1). Instead, this second mutex is granted under the condition that it is available and all threads with lower thread IDs have already acquired such a phase-1 mutex. After the second mutex acquisition, the thread enters phase-2, in which a mutex request suspends a thread as in PDS-1. A new round is started as soon as all threads are suspended.

In both PDS algorithms, the number of threads is constant during the execution of a round. New threads may be created or removed only at the start of a new round. Even then, a deterministic rule for changing the set of threads is necessary. The state of the incoming message queue cannot be used for deciding an adjustment of the thread pool size, as the group communication system only ensures a consistent order of message reception, but no consistent time (i.e., some replica might already have received a message m, while other replicas have not). The PDS algorithms work best if all threads repeatedly execute lock requests followed by computations of approximately identical computation times. It requires no communication for deterministically assigning mutex locks to threads. The algorithm has two main disadvantages. First, as long as one single thread fails to request a mutex lock, no new round can be started. Second, the number of threads must be known deterministically at the start of each round. Incoming requests have to be mapped to a fixed-size thread pool. This means that in each round, new requests have to be assigned to threads that have finished executing their previous requests. If no new requests are available, the system cannot start a new round (as the idling thread will not acquire a lock). The only way to solve this problem is to deterministically create artificial requests in case that client requests do not arrive sufficiently frequently.

4 Extending LSA and PDS

In this section, we define extended versions of the LSA and PDS algorithms. Their primary goal is to support a system model that includes the following features: reentrant locks, nested invocations, condition variables, and time-bounded wait operations. The ADETS-SAT and ADETS-MAT algorithms already support all of these features. The semantics for condition variables is based upon the native Java programming language. In addition, we extend LSA to support an arbitrary number of mutexes without prior registration at the scheduler.

Reentrant locks can be implemented on the basis of any deterministic scheduling algorithm that supports just simple locks. A reentrant mutex is one that can be acquired multiple times by one thread. The transformation requires a data structure that, for each thread, stores the number of times a lock has been acquired. Only on the transitions from 0 to 1 (upon lock) and 1 to 0 (upon unlock), the lock/unlock functions of the base algorithms are called.

4.1 Extending LSA to ADETS-LSA

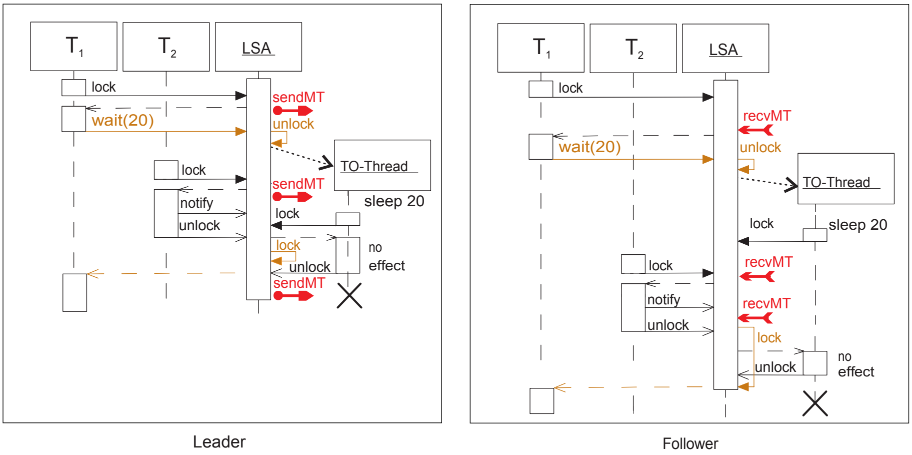
Nested invocations do not require any dedicated support in the implementation of the LSA algorithm. Nested invocations do not influence the order of mutex assignments. In LSA, a thread waiting for a nested invocation reply does not have any influence on the progress of other threads.

Condition variables without time bounds cause no problems in LSA. A `wait()` operation can be called locally at all replicas. Invocations of `wait()` and `notify()/notifyAll()` on the same condition variable have to be done in the same relative order. A deterministic order of such concurrent operations is easily obtained in all replicas by the LSA algorithm, as all operations on a condition variable are protected by the acquisition of the corresponding mutex. The basic LSA algorithm guarantees a deterministic order of these mutex acquisitions. Hence, the order of operations on the condition variable will be deterministic as well.

Time bounds on wait operations represent a more difficult source of non-determinism. For example, two threads T_1 and T_2 might be waiting on a condition variable, with thread T_1 having specified a time bound. A third thread, T_3, might call a `notify()` operation. The timeout of T_1 and the notification of T_3 happen concurrently; thus, the order in which the two happen is non-deterministic. Two possible execution sequences are (a) that the timeout happens first, with the effect that T_1 is resumed by the timeout and, after that, T_2 is resumed by T_3's notify operation, and (b) that T_3's notification happens first, which cancels the timeout and resumes only T_1.

Handling such timeouts deterministically requires a non-trivial extension to LSA. In the solution that we provide in the ADETS-LSA algorithm, a local timeout of a wait operation does not resume the waiting thread directly. Instead, it creates a new thread, which is also subject to the ADETS-LSA scheduling. The thread tries to resume the waiting thread by locking the corresponding mutex and signalling the `wait()` operation to resume. Thus the basic scheduling algorithm guarantees that, due to the lock, the signalling is done in a consistent order on all replicas.

A sample execution of this extension is shown in Figure 1. Thread T_1 calls `wait()` with a timeout of $20ms$. This call causes the LSA scheduler to create a timeout thread (TO-Thread), which sleeps for $20ms$ and then tries to resume the wait. Concurrently, thread T_2 tries to call `notify()`. Both T_2 and TO-Thread



Leader

Follower

Fig. 1. Sample execution of timeout handling in ADETS-LSA

need to lock the same mutex. On the leader node, T_2 is faster, causing the `notify()` operation of T_2 to resume T_1, and the timeout thread has no effect. On the follower node, the timeout thread requests the lock first, but the LSA scheduler ensures that the lock is first assigned to T_2, resulting in a deterministic behaviour.

The original LSA algorithm assumes that globally known IDs for mutexes and for threads exist. *Basile et al.* describe a method for dynamically adding new mutexes and new threads by explicitly notifying the scheduler. Adding new threads is feasible in practice if the middleware infrastructure controls the creation of threads, as it can notify the scheduler. Mutexes, however, are not created explicitly. In Java, every object can be used as a mutex, and there are no globally consistent IDs for these objects. In ADETS-LSA, the leader replica assigns new mutex IDs automatically on the first lock operation on a not yet known mutex. Follower replicas instead suspend a thread upon a lock operation with an unknown ID. On all replicas, the lock operation can uniquely be identified by the thread ID, as the same thread will lock the corresponding mutex on all replicas. The leader sends its mutex ID with its periodic mutex table broadcast, which enables the follower replicas to learn the new mutex ID.

4.2 Extending PDS to ADETS-PDS

The PDS algorithm first raises the question of *assigning requests to threads.* The algorithm assumes that a thread pool of a fixed size is given. It does not allow the asynchronous creation of new threads for each incoming client request. The original publication simply assumes that sufficiently many requests arrive, so that all threads can continuously execute, without specifying a strategy for assigning requests to threads. In a practical middleware infrastructure, however, such a strategy needs to be implemented. The assignment of requests to threads must be made consistently in all replicas. We suggest two possible strategies:

- A *round-robin strategy* assigns incoming requests to all threads such that, given a thread-pool size of N, the i-th incoming request is assigned to thread $i \bmod N$. This strategy works fine if requests have identical computation times.
- In a *synchronized request assignment strategy*, a thread that has just finished processing of its last request locks the mutex of the incoming message queue. This mutex lock is granted consistently in all replicas, because this operation is also under the control of the PDS, and thus each request is assigned to the same thread in all replicas. Our current implementation uses this strategy.

Nested invocations have no impact on the order of lock assignments and thus are uncritical for consistency, but they can have a serious impact on performance. We propose two different strategies:

- First, nested invocations can be used simply without any support by the scheduling algorithm. In this case, however, a thread that waits for a nested invocation can block all other threads from starting a new round. This approach seems favourable if the duration of the nested invocation is short compared to the execution time between two mutex locks. This approach is used in the following experimental evaluation.

- Alternatively, the scheduler can consider a thread that has issued a nested invocation to be suspended. This enables all other threads to continue executing rounds, but requires a deterministic strategy to resume the thread. For example, if the reply message is processed within some round, the suspended thread can be scheduled for being resumed in the next round. This approach adds an additional delay to each nested invocation; thus, it is not useful with nested invocations that have short duration, like it is the case in our evaluation.

From a consistency point of view, *condition variables* can be supported in the PDS algorithm without much effort. All operations on condition variables are protected by mutex locks, and thus the relative order of these operations is deterministic on all replicas. The only requirement is that a `notify()` operation selects the thread to resume deterministically. This determinism is not guaranteed by the native Java notification mechanisms. By implementing a queue of waiting threads that is modified deterministically by each `wait()` operation, such determinism can easily be achieved though.

The integration into the round execution model is done in the following manner. Once a thread t calls `wait()` in some round, it is considered suspended. After a new round has been triggered, t is removed from the set of active threads. Consequently, the scheduling decisions in the subsequent rounds are done without t. When another thread calls `notify()` during some round, thread t is resumed immediately, but has first to aqcuire the corresponding lock. This lets thread t wait until the start of the next round.

Figure 2 shows a sample execution of two replicas A and B. On replica A the `wait()` operation of thread t_1 happens before the second mutex acquisition of

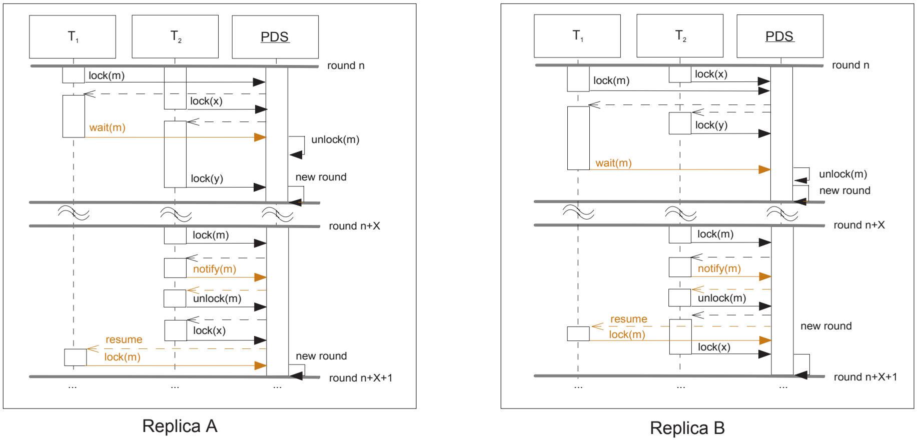


Fig. 2. Handling of condition variables in ADETS-PDS

t_2. Thus, the new round is triggered because of t_2. On replica B the opposite happens. The new round is triggered because of the `wait()` operation.

Having a thread pool of fixed size, however, the use of condition variables can cause deadlocks. If all available threads suspend in `wait()` operations, no more threads are available for handling requests that could resume a waiting thread. To avoid this problem, we implement a strategy for an automated adjustment of the thread-pool size. The original PDS algorithm supports changing the set of threads at the start of a new round. In a deadlock situation, the conditions for the start of a new round (i.e., all threads are blocked) are met. Thus, at the start of each round, the number of threads not blocked in a `wait()` operation is compared to a minimum threshold. If the number falls below the threshold, additional threads are added to the thread pool. On the other hand, if there are more non-waiting threads than the threshold and there are insufficient incoming requests (i.e., the request assignment strategy has to suspend a thread temporarily due to the lack of requests), the number of non-waiting threads is reduced to the minimum threshold.

Timeouts of *time-bounded wait operations* potentially occur concurrently with explicit notifications, and thus an extended algorithm has to make sure that any such non-determinism is avoided. We propose the same concept that is also used for ADETS-SAT and ADETS-MAT. After a timeout occurs, a timeout message is sent to all replicas via group communication. This message is handled by a normal request-handler thread, which notifies the waiting thread. As all notifications are synchronized by mutexes, a deterministic order is guaranteed.

5 Experimental Evaluation

This section presents an experimental evaluation of the scheduling strategies discussed in the previous sections. A set of benchmarks capture typical interaction patterns of distributed applications. Each of them is executed with purely sequential scheduling and with all four multithreaded ADETS variants.

5.1 Implementation Overview

All presented strategies have been implemented on the basis of our FT*flex* replication infrastructure [17], which extends the CORBA-based Aspectix middleware [16,17]. FT*flex* supports multithreading in object replicas using its configurable ADETS (Aspectix DEterministic Thread Scheduler) module. Each scheduling algorithm is implemented as a separate ADETS plug-in module.

Integrating the scheduling module in the middleware is relatively light-weight. We added it in between the group communication module, which delivers the requests, and the object adapter whose task is to enforce at-most-once semantics and to trigger dispatching and parameter unmarshalling. Thus, the scheduler instances are completely independent of the object implementation. We use code transformation to intercept calls of synchronisation-related operation in the object implementation [6].

At runtime, the group communication module receives a new request and passes it on to the scheduler instance. There, a thread that executes the request is created eventually; the creation happens according to the strategy implemented by the scheduler. As soon as the thread is running, the scheduler invokes the object adapter. If the execution of the requested method issues a lock/unlock operation or an operation on a condition variable, these calls are forwarded to the scheduler, which in turn handles these operations. The scheduler itself uses the group communication module to broadcast messages to the other replicas. Such broadcasts might be timeout messages after a time-bounded operation or update messages from the primary in case of the LSA scheduling strategy.

5.2 Benchmark Overview

The benchmarks cover three different scenarios: evaluation of local computations with lock-protected shared state, local computation together with nested invocations, and usage of condition variables. A final discussion analyses overall advantages and disadvantages.

All measurements presented in this section were made on a set of PCs with a AMD Athlon 2.0 GHz CPU and 1 GB RAM. The PCs were using Linux kernel 2.6.17 and were connected by a 100 MBit/s switched Ethernet. The current prototype of the Aspectix middleware was used on the basis of Sun's Java runtime environment version 1.5.0_03.

In each benchmark, active replicas of an object were placed on three nodes, and all clients on separate nodes are started simultaneously in each experiment. All measurements show the invocation times measured at the client side, averaged over at least 5,000 invocations; to minimize the effects of JIT compilation, the first 200 invocations of each client are not included in the average. In all benchmarks the size of the thread-pool in PDS was equal to the number of clients.

5.3 Local Computations

The first group of benchmarks assumes that the behaviour of object methods is limited to (a) performing local computations and (b) requesting and releasing mutex locks. In such a scenario, the only problem of a single-threaded execution is the lack of parallel execution, which primarily is a disadvantage on multi-CPU machines. In the benchmarks, a variable number of clients invoke object methods that have one of the behaviours shown in Figure 3. The measurements for the invocations were made on the client-side.

(*a*) compute
(*b*) compute – lock – state access – unlock
(*c*) lock – state access and compute – unlock
(*d*) lock – state access – unlock – compute

Fig. 3. Variants of the local computations benchmark

The pattern (a) does not access the shared object state and thus does not need any mutex access. The pattern (b) first computes and then locks a mutex, updates the object state, and unlocks the mutex again. This is a typical pattern for applications that first perform computations on the request arguments such as verifying digital signatures and preprocessing the client data, and then use this data to update the object state, using a mutex lock to synchronize the update. Pattern (c) is typical for applications that require simultaneous access to client arguments and object state for performing some calculations. The whole request execution is protected by a mutex lock. Pattern (d) can mainly be found in practice for methods that read the shared state and then perform computations (e.g., transformations of state data) to produce the return value for the client.

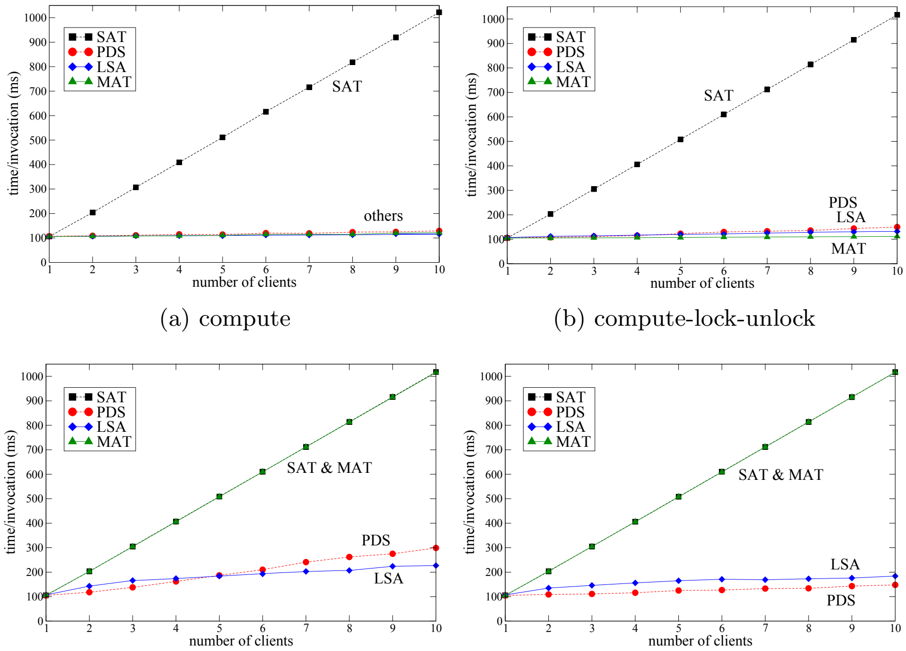
For the following measurements, it is assumed that the local computations take 100 ms. The availability of an unlimited number of CPUs on a single-CPU hardware is simulated by suspending the request-handler thread for the duration of the computation time instead of performing real computations, thus freeing the CPU for other threads. Furthermore, it is assumed that the methods of the replicated object use fine-grained locking. If all methods used the same mutex lock, this would result in a sequential execution. Instead, the benchmarks assume that 10 different mutexes are available, with each client invocation using a randomly selected mutex. The actual state access is assumed to take a negligible amount of time. Figure 4 shows the result of the benchmarks executed with three replicas and a variable number of clients.

With pattern (a), SAT executes all requests sequentially, while all other variants allow a fully concurrent execution. MAT and LSA perform best, as they can execute all requests immediately in the absence of any synchronization. PDS shows a slight overhead, because it requires internal synchronization (i.e., mutex locks for the incoming message queue) for assigning requests to threads.

Pattern (b) results in similar results. While SAT processes all requests sequentially, all other variants enable a concurrent execution of the computations. MAT is the superior variant, as LSA requires communication for the mutex locks, while PDS uses additional mutex locks for assigning requests to threads.

Pattern (c) produces different results. As all requests start with a lock operation and do not define internal scheduling points, the MAT algorithm delays all requests until they become primary and as a result serializes all requests, which leads to the same poor performance as the SAT algorithm. LSA and PDS both enable concurrency and show similar behaviour. With an increasing number of clients, the probability that two requests require the same mutex increases. Such a collision delays the start of a new round for the PDS algorithm; thus, with many clients, the LSA algorithm is superior.

Pattern (d) is similar to (c); the only difference is that mutex locks are released before the computation. The PDS algorithm benefits from this behaviour, as a collision between two request delays a new round only for the short duration of the state access, and not for the duration of the computation. As a result, PDS is the most efficient algorithm for this pattern, while LSA is slightly slower, and both SAT and MAT achieve no concurrent execution.



(a) compute

(b) compute-lock-unlock

(c) lock-compute-unlock

(d) lock-unlock-compute

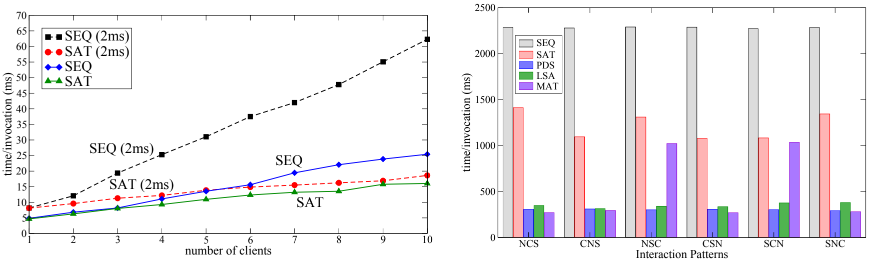
Fig. 4. Measurements with local computations and mutex locks

The different benchmark patterns demonstrate that for each algorithm there are situations in which it performs well, and others in which it does not. Most important, the MAT algorithm is the most efficient one in the situations (a) and (b), while it fails to provide any advantage compared to SAT in the situations (c) and (d). The latter two situations represent worst-case scenarios for MAT. The poor performance of MAT can be alleviated by the introduction of *yield* operations, which enable a selection of a new primary thread without reaching an implicit scheduling point. Another approach for optimizing MAT is the use of source-code analysis to predict synchronization behaviour [19].

5.4 Nested Invocations

The second set of benchmarks adds nested invocations to the patterns. As explained in Section 2, nested invocations can result in deadlocks and reduce performance by causing idle time in a single-threaded execution model. Hence, application patterns with nested invocations are an important scenario even on a single-CPU machine.

In the first scenario, two replica groups A and B are created with each consisting of 3 replicas. A varying number of clients call a method at group A, which in turn calls a method at group B. Internally, both requests and the reply from group B to group A are delivered via group communication.



(a) nested invocations only

(b) nested invocations, local computations, and mutex locks

Fig. 5. Measurements with nested invocations

Figure 5(a) shows the average invocation time measured by the clients, using (i) a strictly sequential execution and (ii) the ADETS-SAT algorithm[1]. The solid lines (diamond and triangle symbols) refer to measurements in which the nested invocation returns immediately. Even in this situation, multithreading with ADETS-SAT is increasingly better with a rising number of clients. In a second measurement (dashed lines with circles and squares), the method called at B suspends for $2ms$ before it returns. In this case, the benefit from our multithreaded approach (which allows to accept new requests at A while the invocation to B is in progress) is enormous compared to a single-threaded execution.

A comparison of all ADETS scheduling algorithms is provided on the basis of a set of more complex benchmarks. In each benchmark, the replicas execute the following operations: a nested invocation ($100 \ldots 150ms$, denoted as N), a local computation ($75 \ldots 125ms$, denoted as C), and a synchronized state update (lock und unlock operation, denoted as S).

The duration of nested invocations and local computations was simulated to have a uniform random distribution on the given interval. The three elements can be combined in six permutations (NCS, NSC, CNS, CSN, SNC, SCN). Figure 5(b) shows the result of the benchmarks with above parameters, run with ten clients.

The ADETS-SAT performs better than the single-threaded execution, because the idle time of a nested invocation is utilized. Local computations cannot be performed in parallel, however. Thus, the ADETS-SAT performs worse than the other algorithms. The performance of the ADETS-MAT algorithm heavily depends on the interaction patterns. In some situations (NCS, CSN), the algorithm performs best of all. In others (NSC, SCN) it offers no significant advantage compared to the ADETS-SAT algorithm. The problematic pattern is a state update (S) followed by a computation (C). The ADETS-PDS performs

[1] No other algorithms have been evaluated in this nested-invocation-only benchmark. As there are no lock operations, ADETS-MAT and ADETS-LSA would result in similar performance as ADETS-SAT. On the other hand, ADETS-PDS, which assumes that each thread acquires a lock in each round, would not work appropriately.

well in all interaction patterns and performs even better than ADETS-LSA. The performance of neither of them significantly depends on on the pattern.

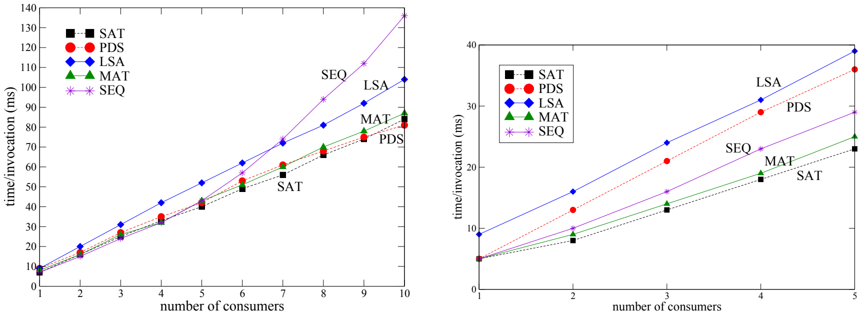
5.5 Condition Variables

Condition variables are an important mechanism that enables a request to wait for another request. To examine the performance of the algorithms in combination with condition variables, we evaluated two scenarios, an unbounded buffer scenario and a bounded buffer scenario.

A replicated object that implements the *unbounded buffer* provides two methods, `consume()` and `produce()`. The `consume()` method returns an available data item on a condition variable if no item is available. The `produce()` method makes an item available, and notifies another request-handling thread that waits on the condition variable, if such a thread exists. Without support for condition variables, the consume method needs to be implemented differently; for the evaluation we use periodic polling for `consume()` with pure sequential scheduling.

Figure 6(a) shows the result of this experiment, using a single producer client and up to ten consumer clients. With an increasing number of consumers, the single-threaded execution shows an increasing disadvantage. This behaviour is to be expected due to the periodic polling: the number of unsuccessful iterations of `consume()` calls increases with a rising number of consumers competing for the producer. The other strategies, however, scale linearly because a thread is only notified if an item in the buffer exists. The ADETS-SAT performs minimally better than ADETS-MAT and ADETS-PDS. The ADETS-LSA, however, has a notable overhead due to the leader-follower communication.

The second benchmark for evaluating the scheduler behaviour in combination with condition variables implements a *bounded buffer*. In this experiment, both `produce()` and `consume()` block if the buffer is full or empty, respectively. Two condition variables are used: the first one is used to resume a blocked `produce()` call by a `consume()` call; the second one is used in the reverse direction. Figure 6(b) shows the result of the experiment, in which the same number of producers



(a) unbounded buffer

(b) bounded buffer

Fig. 6. Measurements with condition variables

and consumers, each ranging from 1–5, have been used. The size of the buffer was set to 2. The graph shows the average time per consumer invocation; exactly the same average time was obtained for producer invocations.

Both experiments show that ADETS-SAT and ADETS-MAT are superior to all other execution strategies. ADETS-PDS and ADETS-LSA, on the other hand, show poor performance. In the experiment with the bounded buffer, they perform even worse than the sequential polling-based approach. With the ADETS-LSA algorithm, this is due to the additional communication caused by the scheduling algorithm. With ADETS-PDS, threads that resume from a wait operation need to be delayed until the next internal round starts; this delay increases the invocation times.

5.6 Overall Performance Comparison

Concluding, we state that the ADETS-SAT algorithm always performs better than the single-threaded execution. On the other hand, the main disadvantage is that it does not support true multithreading. As a result, when using multiple CPUs it yields significantly less performance than to the other strategies.

The ADETS-MAT algorithm in contrast supports true multithreading in typical usage patterns, such as preprocessing some data and then modifying the local state, but it does not perform well if calculations require a locked mutex, because different mutexes cannot be locked concurrently by different threads.

The ADETS-LSA strategy works well independent of a certain pattern. The leader-follower communication, however, is one disadvantage of the ADETS-LSA. Obviously, this issue is even more important when using the LSA in a WAN environment. Due to the communication, the ADETS-LSA also showed a noticeable overhead in scenarios with condition variables. A second disadvantage is the reconfiguration process that is necessary after the failure of the leader. This failure must first be detected, which leads to a delay until the reconfiguration process can be started. Furthermore, this reconfiguration defeats essential advantages of active replication, as it is typically used if minimal downtime after failures is required.

The ADETS-PDS has a good overall performance when all threads execute the same pattern concurrently. But it does not perform well in other scenarios when different patterns are executed, because the round execution model can cause high delays. Also, in conjunction with condition variables the PDS may perform worse than ADETS-SAT and ADETS-MAT.

6 Conclusions

In this paper, we have revisited the problem of multithreaded execution of methods at replicated objects. In active replication, multithreading is a potential source of non-determinism that has to be made deterministic by an adequate thread-scheduling strategy. Similar consistency problems can arise in passive replication if the re-execution of methods after a primary failure is inconsistent to the first execution of the primary before its failure. In our CORBA-based FT*flex* infrastructure for object replication, we have implemented four different strategies: ADETS-MAT, ADETS-SAT, ADETS-LSA, and ADETS-PDS.

The first contribution of this paper are the ADETS-LSA and ADETS-PDS algorithms, an extension of Basile's LSA and PDS algorithms. Our variants add support for the native Java synchronization model. Beside reentrant locks, the most important extensions are the support for condition variables and the deterministic handling of time bounds on wait operations. PDS was also extended to support nested invocations.

The second contribution is a comparison of the available thread-scheduling strategies. In a set of experiments, we have analysed the respective benefits of each algorithm. We show that the performance of an algorithm is highly dependent on the interaction patterns. No algorithm is clearly superior to all others. Our evaluation provides information about which algorithm works better in which application scenarios. We conclude that replication infrastructures should support variability of its thread-scheduling strategy. In all cases, however, a multithreaded strategy is superior to single-threaded request execution.

References

1. OMG: Common object request broker architecture: Core specification, version 3.0.3. Object Management Group (OMG) document formal/2004-03-12 (2004)
2. Montresor, A.: The Jgroup distributed object model. In: Proceedings of the IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems II (1999)
3. Zhao, W., Moser, L.E., Melliar-Smith, P.M.: Deterministic scheduling for multithreaded replicas. In: WORDS 2005 (2005)
4. Basile, C., Whisnant, K., Kalbarczyk, Z., Iyer, R.: Loose synchronization of multithreaded replicas. In: SRDS 2002 (2002)
5. Basile, C., Kalbarczyk, Z., Iyer, R.: Preemptive deterministic scheduling algorithm for multithreaded replicas. In: DSN 2003 (2003)
6. Domaschka, J., Hauck, F.J., Reiser, H.P., Kapitza, R.: Deterministic multithreading for Java-based replicated objects. In: PDCS 2006 (2006)
7. Reiser, H.P., Hauck, F.J., Domaschka, J., Kapitza, R., Schröder-Preikschat, W.: Consistent replication of multithreaded distributed objects. In: SRDS 2006 (2006)
8. Felber, P., Guerraoui, R., Schiper, A.: The implementation of a CORBA object group service. Theory and Practice of Object Systems 4(2), 93–105 (1998)
9. Bessani, A.N., da Silva Fraga, J., Lung, L.C., Alchieri, E.A.P.: Active replication in CORBA: Standards, protocols, and implementation framework. In: DOA 2004 (2004)
10. Fich, F.E., Hendler, D., Shavit, N.: On the inherent weakness of conditional synchronization primitives. In: PODC 2004 (2004)
11. Napper, J., Alvisi, L., Vin, H.: A fault-tolerant Java virtual machine. In: DSN 2003 (2003)
12. Friedman, R., Kama, A.: Transparent fault tolerant Java virtual machine. In: SRDS 2003 (2003)
13. Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., Zainlinger, R.: Distributed fault-tolerant real-time systems — the Mars approach. IEEE Micro 9(1), 25–40 (1989)

14. Poledna, S., Burns, A., Wellings, A.J., Barrett, P.: Replica determinism and flexible scheduling in hard real-time dependable systems. IEEE Trans. Computers 49(2), 100–111 (2000)
15. Jiménez-Peris, R., Patiño-Martínez, M., Arévalo, S.: Deterministic scheduling for transactional multithreaded replicas. In: SRDS 2000 (2000)
16. Reiser, H.P., Hauck, F.J., Kapitza, R., Schmied, A.I.: Integrating fragmented objects into a CORBA environment. In: Proc. of the Net.ObjectDays, Erfurt, Germany (2003)
17. Reiser, H.P., Kapitza, R., Domaschka, J., Hauck, F.J.: Fault-tolerant replication based on fragmented objects. In: Eliassen, F., Montresor, A. (eds.) DAIS 2006. LNCS, vol. 4025, pp. 256–271. Springer, Heidelberg (2006)
18. Narasimhan, P., Moser, L.E., Melliar-Smith, P.M.: Enforcing determinism for the consistent replication of multithreaded CORBA applications. In: SRDS 1999 (1999)
19. Domaschka, J., Schmied, A.I., Reiser, H.P., Hauck, F.J.: Revisiting deterministic multithreading strategies. In: Int. Workshop on Java and Components for Parallelism, Distribution and Concurrency (2007)