Consistent Replication of Multithreaded Distributed Objects

Hans P. Reiser, Jörg Domaschka, Franz J. Hauck
Distributed Systems Lab, Faculty of Computer Science
University of Ulm, Germany
{hans.reiser, joerg.domaschka, franz.hauck}@uni-ulm.de

Rüdiger Kapitza, Wolfgang Schröder-Preikschat
Department of Distributed Systems and Operating Systems
University of Erlangen-Nürnberg, Germany
{kapitza, wosch}@informatik.uni-erlangen.de

Abstract

Determinism is mandatory for replicating distributed objects with strict consistency guarantees. Multithreaded execution of method invocations is a source of nondeterminism, but helps to improve performance and avoids deadlocks that nested invocations can cause in a single-threaded execution model. This paper contributes a novel algorithm for deterministic thread scheduling based on the interception of synchronisation statements. It assumes that shared data are protected by mutexes and client requests are sent to all replicas in total order; requests are executed concurrently as long as they do not issue potentially conflicting synchronisation operations. No additional communication is required for granting locks in a consistent order in all replicas. In addition to reentrant mutex locks, the algorithm supports condition variables and time-bounded wait operations. An experimental evaluation shows that, in some typical usage patterns of distributed objects, the algorithm is superior to other existing approaches.

1 Introduction

Object replication can be used to build reliable object-based distributed applications. Active and passive replication are two basic strategies for managing the state of replicas. In passive replication, a primary replica executes all client requests and transfers its state to the secondary replicas. In active replication, all replicas execute all method requests independently. If the replicas have an identical initial state and deterministic behaviour, they maintain a consistent state. In general, concurrent client requests are processed in a consistent order in all replicas by distributing them with an atomic multicast protocol.

Many existing object-replication systems use a single-threaded request-execution model, as the scheduling of multiple concurrent threads is a source of nondeterminism. A strictly sequential execution of requests, however, reduces the performance, can cause deadlocks, and limits the types of synchronisation that can be used (see Section 2).

The main contribution of this paper is the novel ADETS-MAT (*Aspectix DEterministic Thread Scheduler – Multiple Active Threads*) algorithm for scheduling threads in replicated objects. It guarantees that synchronisation operations are executed in a deterministic order. ADETS-MAT enables the concurrent execution of multiple threads in replicated objects. Furthermore, the algorithm 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.

This paper focuses on active replication. Nevertheless, our work is also relevant for passive replication. In that replication style, the state transfer is often not triggered immediately after each state modification. A new primary may not have the most recent state that the previous primary had before crashing. The new primary can use a request log to re-execute all operations that have been processed after the last state transfer. A state identical to that of the failed primary is only reached if executing this sequence of operations is deterministic.

We claim that the ADETS-MAT algorithm performs well in some typical usage patterns of a distributed object. One example of such a typical pattern is a method that first performs local computations on the method arguments, and then acquires a mutex lock, modifies the object state, releases the lock, and returns. Only one thread can enter the state modification section at a time; however, the preceding phase of local computations can be executed concurrently in multiple threads.

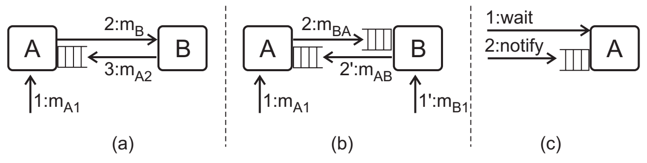

Figure 1. Thread-execution problems

The remaining paper is structured as follows. The next section discusses the abilities and problems of existing approaches. Section 3 describes our system model and the basic idea of the approach. Section 4 presents our algorithm. Section 5 proves the correctness of our algorithm. Section 6 gives an evaluation. Finally, Section 7 concludes.

2 State of the Art

Depending on the thread-execution model, a servant implementation may face several problems, which we classify into the following categories:

NESTEDDEADLOCK: Let's assume that an object A, while executing some method m_{A1}, issues a remote method m_B at another object B, and method m_B in turn invokes a method m_{A2} at object A. If the object A operates strictly sequential, it will not handle m_{A2} before m_{A1} returns. However, m_{A1} will only return after m_B (and, consequently, m_{A2}) has returned; this results in a deadlock (see Fig. 1a).

MUTUALDEADLOCK: Let's assume that an object A executes a method m_{A1}, which invokes a remote method m_{BA} on object B. Furthermore, let us assume that in parallel the object B executes a method m_{B1}, which invokes a remote method m_{AB} at object A. Both the invocations of m_{AB} and m_{BA} cannot be handled in a strictly sequential execution, as both objects will execute these invocations only after m_{A1} and m_{B1}, respectively, will have returned. Again, a deadlock is reached (see Fig. 1b).

NOCONDITIONWAIT: With condition variables, a thread can suspend until it is notified by another thread. Many replication infrastructures do not support this programming construct. If an object implementation based on condition variables were used in a single-threaded execution model, it would deadlock (see Fig. 1c). In such situations, workarounds such as periodic polling have to be used, which usually have higher overhead than simply waiting on a condition variable.

NESTEDIDLING: Let's assume that an object A executes a method m_{A1}, which invokes a method m_B on object B. In a single-threaded execution model, object A will not process any other request before m_B returns and m_{A1} finishes. Such idling during nested invocations is less efficient than handling the next invocation with a second thread while the first thread waits for the nested-invocation reply.

NOPARALLELISM: Modern computer architectures often include multiple CPUs or multi-core CPUs, allowing true parallel execution of requests. If the replication infrastructure does not allow the truly parallel execution of multiple requests, it cannot efficiently use the benefits from such architectures.

EXPLICITSYNC: If multiple threads may execute concurrently, access to shared object state has to be coordinated. For this purpose, the developer has to provide explicit synchronisation statements (such as mutex locks), which serialise concurrent modifications.

Existing object replication systems use thread-execution models that can be classified into the following categories:

SEQUENTIAL:In a sequential execution, a request is only processed after the preceding request has been completed. This single-threaded model is widely used in fault-tolerant middleware systems (e.g. OGS [5], GroupPac [6]). Given a total order of all incoming requests and deterministic replica behaviour, consistency is easily obtained. This approach does not require explicit synchronisation of state access; it suffers from all problems mentioned above except EXPLICITSYNC.

SINGLELOGICALTHREAD (SLT): In this model, a single logical thread of execution exists. This logical thread may nestedly call methods of the same object multiple times. For example, in Figure 1a this means that the replica A detects that the invocation m_{A2} belongs to the same logical thread as m_{A1}, permitting the execution of m_{A2}. This way, the NESTEDDEADLOCK problem is removed; because of the single logical thread, no explicit synchronisation is needed. Technically, context information that identifies the originating logical thread is propagated through remote call chains. If an object receives a request that belongs to the current logical thread, it executes this request with an additional physical thread. No inconsistencies can arise, as the first thread remains blocked in all replicas during the execution of the nested invocation, and only resumes after the additional physical thread has finished. Such a model was first used in the Eternal system [12].

SINGLEACTIVETHREAD (SAT): In this model, multiple physical threads can exist within a replica, with only one of them being active at a time, and all others being blocked (e.g., waiting for a lock or for a nested invocation). If the active thread blocks or terminates, a deterministic strategy is used to resume one of the existing threads or to create a new active thread for handling the next request. An algorithm using this model was first suggested by Jimenez-Peris et al. [9] for a transactional, conversational client-server interaction model. Zhao et al. [14] proposed a similar model in a simpler RPC-based replicated object.

MULTIPLEACTIVETHREADS (MAT): In this category, multiple threads may exist and be concurrently active. Only in this model can multiple threads within a replicated object

	SEQUENTIAL	SLT	SAT	MAT
CIRCULARDEADLOCK	×			
MUTUALDEADLOCK	×	×		
NOCONDITIONWAIT	×	×		
NESTEDIDLING	×	×		
NOPARALLELISM	×	×	×	
EXPLICITSYNC			×	×

Figure 2. Execution models and problems

benefit from multiple CPUs or a multi-core CPU. To maintain consistency, all access to shared data structure needs to be made in a consistent order. Two algorithms for this model have previously been suggested by Basile et al. [2–4]. We compare them to our algorithm in Section 6.

Figure 2 correlates the four models with their capability to solve the aforementioned problems. Only SEQUENTIAL and SLT avoid the `ExplicitSync` problems. Only algorithms in the MAT category can handle all other problems.

The problem of deterministic replication of multithreaded objects may also be addressed below the middleware level. For example, Friedman et al. [7] use a modified JikesRVM to achieve consistent thread scheduling for replicated Java objects; a similar approach, based on a modified Sun JDK 1.2, is suggested by Napper et al. [11]. Other systems approach the problem at even lower system levels. For example, MARS [10] is strictly time-driven and periodic at a hardware level, which makes all functional and timing behaviour strictly deterministic. The features of such a platform can be used for deterministic replication [13]. The drawback of such systems is that they all require support in hardware, operating system, or Java virtual machine. In contrast, our work assumes an asynchronous system model and provides deterministic functional behaviour of multithreaded replicated objects purely at the middleware level.

3 System model and basic concept

We assume that a set of identical object replicas is located on different nodes and connected via a network. Clients interact with the replicated object by remote method invocations. Each client request creates a new thread in all replicas. All threads may concurrently modify the object state. Access to shared data is synchronised by mutexes.

Replica implementations can invoke nested invocations on other replica groups. The replication infrastructure makes sure that a single invocation is made jointly for all replicas, and then propagates the invocation reply to all replicas by totally ordered multicast.

We assume a synchronisation model such as that of the Java programming language. The number of mutexes is not limited. A thread may incrementally acquire an arbitrary set of mutexes; mutexes are reentrant, that is, they can be acquired multiple times by the same thread. Any mutex is associated with a single condition variable that allows a thread to wait for a notification. Invoking a wait operation requires the prior acquisition of the associated mutex, which is released during the wait and re-acquired as soon as the thread subsequently resumes. Similarly, notifying a waiting thread requires prior mutex acquisition. In addition, wait operations can be limited by time bounds.

The state transitions of an object are assumed to be deterministic given a specific order of mutex assignments and incoming messages. We divide the execution into *thread execution intervals* (see Definition 1) and use these intervals to define *piecewise thread determinism* (see Definition 2).

Definition 1 (Thread Execution Intervals) *A scheduling point s_i of a thread t is defined by any of the following activities of t: thread creation, request of a mutex lock, wait request on a condition variable, nested invocation, and thread termination. An execution interval e_i of a thread is the activity of a thread between s_i and s_{i+1}.*

Thread creation always defines the first scheduling point s_0, and thread termination defines the last scheduling point s_N. The scheduling points $s_k, 0 < k < N$ may temporarily suspend the thread waiting for a mutex, for a condition variable, or for a nested invocation; the next execution interval e_k is started as soon as the lock is granted, the wait operation is notified or has timed out, or the reply for the nested invocation arrives, respectively. If the ADETS-MAT algorithm is extended by an explicit `yield` operation (see Section 6), this operation also defines a scheduling point. The behaviour at the scheduling points is defined by the multithreading algorithm; determinism at these points is not subject to the replica implementation. Between the scheduling points (i.e., during an execution interval), the object replica implementation is required to be piecewise deterministic.

Definition 2 (Piecewise Thread Determinism) *Let $L_r(t)$ be the local state of a thread r, and $S_{r,i}(t)$ be the part of the shared object state that thread r can access in execution interval e_i based on previous lock operations. A thread r is piecewise deterministic iff the local state $L_r(t_a)$ and the protected part of the shared state $S_{r,i}(t_a)$ at the beginning of e_i uniquely define the state of $L_r(t_b)$ and $S_{r,i}(t_b)$ at the end of the execution interval e_i.*

The initial local state of a thread is defined by the request message that created the thread. After a nested invocation, it is the local state at the invocation time plus the invocation reply. In all other cases, the local state at the start of execution interval e_i is equal to the state at the end of e_{i-1}.

The state of $S_{r,i}(t_a)$ at the beginning of an execution interval e_i depends on the sequence of threads that previously had access to parts of this state. Assuming piece-

ActivePrimary:	ThreadID
LockedMap:	Map<Obj,[ThreadID,count]>
MutexWaitMap:	Map<Obj, Queue<[ThreadID,count]>>
CondWaitMap:	Map<Obj, Queue<[ThreadID,ID,count]>>
PrimCandidates:	Queue<[ThreadID, Queue<Action>]>
CurActionList:	Map<ThreadID, ref to Queue<Action>>

Figure 3. Data structures of ADETS-MAT

wise deterministic behaviour of each thread execution interval that previously modified the shared state, the order of these modifications is essential to guarantee consistency of the shared state.

The system is assumed to be asynchronous; no strict bounds for the duration of computation or for communication delays exist. It is assumed that all shared data is protected by mutex locks. The order in which concurrent threads try to acquire mutexes is non-deterministic. Arbitrary client requests can arrive at the replicas with an unknown, varying delay. Similarly, replies to nested invocations may arrive at an unknown time. The execution speed of concurrent threads is also non-deterministic and may vary between replicas. Therefore, no *a priori* definition of a lock acquisition order is possible. It is the purpose of a deterministic multithreading algorithm to ensure an identical ordering in spite of a potentially concurrent execution of the threads. We assume that the middleware is able to intercept all synchronisation operations of the replicas.

The basic idea of the ADETS-MAT algorithm is as follows: We divide the set of existing threads into *primary* and *secondary* threads. Only one of the primary threads is executing at a time. All other primary threads are suspended, which means that they are waiting for (a) a mutex lock or (b) a condition variable notification. If the active primary thread terminates or suspends, the scheduler tries to resume a suspended primary thread. If no resumable primary thread exists, a deterministically selected secondary thread T_i is promoted to be the active primary thread, giving it the ability to perform synchronisation-related operations. All secondary threads may run in parallel to the primary thread, as long as they make no actions that interfere with the scheduling of primary threads. If a secondary thread requests a lock or wants to wait on a condition variable, it is suspended until it becomes primary. If it releases a lock or issues a notify operation, the thread may continue, but the actual operation is deferred until the thread becomes primary. Our algorithm does not provide fair scheduling. A running primary thread is not preempted; if it does not suspend or terminate, it will prevent all other threads from acquiring a lock. In Section 6, we will discuss extensions that reduce this problem.

4 The ADETS-MAT Algorithm

The ADETS-MAT algorithm allows multiple threads to run concurrently within a single object. We assume that different replicated objects on the same node are independent from each other. Scheduling is done on a per-object basis, and each replica uses its own instance of the scheduling algorithm. An object implementation protects all access to common state variables by mutex locks, and the implementation is piecewise deterministic as defined in Section 3. ADETS-MAT supports reentrant locks, Java-style condition variables (i.e., each mutex has an associated condition variable, on which the application calls `wait`, `notify` and `notifyAll` operations); threads blocked in a `wait` operation can be unblocked by a timeout.

4.1 Data Structures

Figure 3 shows the essential data structures that our algorithm uses. The term `Obj` is used to refer both to a mutex and to a condition variable; this implies the assumption that for each mutex there exists exactly one condition variable.

`ActivePrimary` specifies the currently active primary thread. Only the active primary thread may acquire or release locks or modify the list of threads waiting for a lock or condition variable. An arbitrary number of additional secondary threads can run in parallel, but these threads may not influence the lock acquisition order.

`LockedMap` is used to store the information about which mutex is locked by which thread. Reentrant locks are supported by a counter which is incremented/decremented on each lock/unlock operation of the same thread. If the counter reaches the value 0, the lock is no longer held by the thread, and the mutex entry is removed from the map. Any mutex not in `LockedMap` is free. Only the active primary thread may add a new entry to `LockedMap` or remove an entry from it.

`MutexWaitMap` stores a list of threads that are waiting for a mutex. Only the active primary thread will be added to this map; after the addition, the thread will suspend and a new primary will be selected deterministically. The value `count` specifies how many times the mutex shall be locked. For all explicit `lock` operations, `count` will be equal to 1. If a thread resumes from a `wait` operation and needs to reacquire the associated lock, `count` is set to the reentrance count that the lock had before the `wait` operation.

`CondWaitMap` stores all threads that are waiting on a condition variable. Identical to `MutexWaitMap`, threads will only be added while they are active primary. In addition to the thread ID, a unique ID is created for each invocation of a `wait` operation. The unique ID is used to correctly assign `Timeout` messages to `wait` operations.

`PrimCandidates` is an ordered queue with an entry

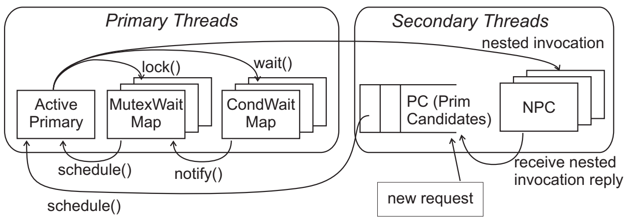

Figure 4. Thread states in ADETS-MAT

for each received message. An entry contains a reference to a secondary thread t_s that handles the message in parallel to the primary thread, and a list of deferred actions (e.g., unlock and notify operations) that t_s requested, but which only may be executed after t_s becomes the primary thread. If t_s performs an action that is not allowed for secondary threads (lock and wait), the thread suspends until it becomes the active primary.

Multiple entries in `PrimCandidates` can reference the same thread. For example, if a thread issues multiple nested invocations while executing as a secondary thread, each reply creates a `PrimCandidate` entry. The `CurActionList` maps the Thread ID to the deferred action list in `PrimCandidates` that corresponds to the current execution (i.e., the last reply that resumed the thread).

4.2 Algorithm

Figure 4 illustrates the states that a thread can have and the possible transitions. A thread belongs to the set of either the *primary* or the *secondary* threads. The *primary* set comprises a single active thread (stored in `ActivePrimary`) and a set of suspended threads that are waiting for a lock or for a condition variable (stored in `MutexWaitMap` and `CondWaitMap`, respectively). *Secondary* threads that are referenced from `PrimCandidates` entries are primary candidates (PC); all other secondary threads are non-primary candidates (NPC). The order in the `PrimCandidates` queue is defined by the total order of messages received from group communication. Each entry in the queue has a reference to a secondary thread. When the queue head is used to select the next active primary thread, the referenced thread may still be running, it may be suspended because of a operation that only the active primary may perform, it may be suspended due to a nested invocation, or it may have terminated.

Any arriving message creates an entry in the `PrimCandidates` queue with a reference to a thread. In case of a client request, this reference points to the new thread, which is created and started to handle the request. If the message is a nested invocation reply, the reference points to the thread waiting for this reply, and this thread is resumed. In case of a `TIMEOUT` message, the thread reference points to the thread that executed the corresponding `wait` operation, and a timeout entry with the UUID from the message is stored in the action list.

A `schedule` operation is responsible for activating a new primary thread. It is called if the current active primary thread suspends or terminates, or if no active primary exists and an entry is added to an empty `PrimCandidates` queue. The method first tries to select a runnable thread from `MutexWaitMap`. If no such thread exists, the queue head of `PrimCandidates` is examined. If the queue is empty, `schedule` terminates; adding the next entry to the queue will call `schedule` again. Otherwise, the head element is removed from `PrimCandidates` and the referenced thread becomes the active primary.

If the primary thread issues a nested invocation, it cannot resume before the corresponding reply arrives. It is removed from the set of primary threads and becomes an NPC member. As soon as the nested invocation reply arrives, the reply message with a reference to the thread is added to `PrimCandidates`; the thread becomes a PC and may resume execution as a secondary thread.

Figure 5 shows a specification of our algorithm in pseudocode. The description assumes the Java synchronisation model, in which an object can be used both as a mutex and as a condition variable. The main component is the `schedule` function implemented in lines 1–28. The function is called (a) when no active primary exists and a new message arrives (line 32), (b) when the current primary thread terminates (line 48), and (c) when it suspends (it issues a nested invocation, line 52; it calls `wait` on a condition variable, line 97; or it calls `lock` on a mutex locked by another thread, line 86).

The `schedule` implementation first examines `MutexWaitMap` for resumable threads that are blocked on a synchronisation operation (lines 3–9). A thread can be resumed if it has requested a mutex lock that is now available (i.e., has no entry in `LockedMap`). This also covers threads that issued a `wait` operation: if a thread in `CondWaitMap` is notified by another thread or a timeout, it is moved from `CondWaitMap` to `MutexWaitMap`, as it has to re-acquire the lock prior to continuation. In case of multiple available mutexes in line 3, the selection must be deterministic, e.g., by using a total order on mutex IDs. If a runnable thread is found, it is resumed (line 8).

If no resumable thread is found, the `PrimCandidates` queue is examined. If the queue is empty, no primary thread is selected, and `schedule` terminates (line 11); it is re-invoked as soon as a new message arrives. Otherwise, `schedule` picks the first element from `PrimCandidates` (line 12) and processes the action list from the queue entry (i.e., executes deferred actions, lines 13–24). If the thread that corresponds to the queue element is blocked due to a `wait` or `lock` call while being sec-

```
function schedule():
    find obj with MutexWaitMap(obj) $\neq$ nil and
        LockedMap(obj) = nil
    **if** obj exists:
        (tid,n) := MutexWaitMap(obj).removeFirst()
        LockedMap(obj) := (tid,n)
        ActivePrimary := tid;
        tid.resume()  // resume suspended thread
        **return**
    **if**(PrimCandidates.isEmpty())
        ActivePrimary := **null**; **return**
    (tid, alist) := PrimCandidates.removeFirst()
    foreach entry in alist:
        **case** TIMEOUT(id):
            **if** [tid, id, count] $\in$ CondWaitMap(obj):
                CondWaitMap(obj).remove([tid, id, count])
                MutexWaitMap(obj).append(tid, count)
                tid := **null**
        **case** TERMINATE, WAIT_NESTED:
            tid := **null**
        **case** WAIT, LOCK:
            // thread is resumed below as ActivePrimary
        **case** UNLOCK(obj)/NOTIFY{|ALL}(obj):
            call primary{Unlock|Notify|NotifyAll}(obj, tid)
    **if**(tid!=**null**):
        ActivePrimary := tid
        **if**(tid is suspended) tid.resume()
    **else** schedule()

function appendPrimCandidate(element):
    PrimCandidates.append(element)
    **if**(ActivePrimary==**null**) schedule()
    CurActionList(element.tid) =
        pointer to element.alist

function receive(message):
    **if** message is new client request:
        tid := **new** thread(message)
        appendPrimCandidate([tid, ()]); tid.run()
    **if** message is TIMEOUT(obj,tid,id)
        Timer.cancel(TIMEOUT(obj,tid,id))
        appendPrimCandidate([tid, (TIMEOUT(id))])
    **if** message is nested invocation reply for thread tid:
        appendPrimCandidate([tid, ()]);
        tid.deliver(message)  // resume thread

On termination of thread tid:
     **if** tid == ActivePrimary: schedule()
     **else** CurActionList(tid).append(TERMINATE)

On nested invocation (Request r) by thread tid:
    **if** ActivePrimary == tid: schedule()
    **if** ActivePrimary $\neq$ tid:
        CurActionList(tid).append(WAIT_NESTED)
    r.invoke()
    suspend until reply is received

function primaryUnlock(obj, tid ):
    [ tid ‘, k] := LockedMap(obj);  assert  tid ‘ == tid
    **if** k>1:  LockedMap(obj) := [ tid , k−1]
    **else** :      remove LockedMap(obj)

function primaryNotify ( obj ):
    remove first element [ thread , id, count] from
        CondWaitMap[obj]
    Timer.cancel (TIMEOUT(obj,thread,id))
    MutexWaitMap(obj).append(thread, count)

function primaryNotifyAll ( obj ):
    **for** all elements [thead$_i$, id$_i$, count$_i$] in
        CondWaitMap(obj):
        Timer.cancel (TIMEOUT(obj,thread$_i$,id$_i$))
        MutexWaitMap(obj).append(thread$_i$, count$_i$)
    remove all elements from CondWaitMap(obj)

// intercepted synchronisation actions actions:
lock( obj) by thread tid :
    **if** not primary:
        CurActionList( tid ). append(LOCK(obj))
        tid .suspend    // until primary
    **if** LockedMap(obj) == [tid , n]:     // reentrant lock
        LockedMap(obj) := [ tid , n+1]; **return**
    **else if** LockedMap(obj) == nil:
        LockedMap(obj) := [ tid , 1]       // grant lock
    **else if** LockedMap(obj) == [tid ‘, n] and tid$\neq$tid‘:
        MutexWaitMap(obj).append(tid, 1)
        schedule ()
        tid .suspend     // until LockedMap(obj) == [tid, ?]

wait( obj , timeout) by thread tid :
    [ tid ‘, k] := LockedMap(obj);  assert  tid ‘ == tid
    **if** not primary:
        CurActionList( tid ). append(WAIT(obj,timeout))
        tid .suspend       // until primary
    id := new unique ID
    remove LockedMap(obj)      // fully release lock
    CondWaitMap(obj).append([tid, id , k])
    schedule ()
    **if** timeout > 0:
        Timer. setup ( timeout , TIMEOUT(obj,tid,id))
    tid .suspend   // until LockedMap(obj) == [tid, ?]

unlock( obj) by thread tid :
    **if** (primary)   primaryUnlock(obj, tid )
    **else** CurActionList( tid ). append(UNLOCK(obj))

cond_notify [ _all ]( obj) by thread tid :
    **if** (primary)   primaryNotify [ All ]( obj)
    **else** CurActionList( tid ).
        append(NOTIFY[_ALL](obj))

Timer. setup ( t , message):
    Schedule sending message via abcast after t ms
Timer. cancel (message):
    Cancel sending message **if** not yet sent
```

Figure 5. The ADETS-MAT Algorithm

ondary, it is resumed. If the thread is not runnable (it has terminated or has issued a nested invocation), `schedule` is called again to repeat the selection of a new primary thread (line 28). Otherwise, it becomes the new primary thread.

Lines 36–45 show the processing of new messages from the group communication system. Three kinds of messages may arrive: client requests, `TIMEOUT` messages, and nested invocation replies. For client requests, a new thread is created. For nested invocation replies, the thread waiting for the reply is resumed. In both cases, an entry is added to `PrimCandidates` with a reference to the thread and an empty action list. For `TIMEOUT` messages, an entry with empty thread reference and an action list containing the `TIMEOUT` messages is added to `PrimCandidates`.

The handling of intercepted synchronisation operations is shown in lines 76–109. For `lock` operations, a thread that is not primary has to suspend until it becomes primary (the suspension is recorded in the action list of the `PrimCandidates` entry that will make the thread primary). As soon as the current thread is primary, it tries to acquire the lock. If `lock` is called for an already acquired mutex, only the reentrance count is increased (line 81). If the mutex is free, the lock is granted by putting the thread into `LockedMap` (line 83). If it is locked by another thread, the primary thread creates an entry in `MutexWaitMap`, calls `schedule`, and suspends (lines 84–87).

A `wait` operation suspends any secondary thread until it becomes primary (lines 91–93). Next, the thread is put into `CondWaitMap`, calls `schedule`, and suspends. If a timeout for `wait` is given, the emission of a `TIMEOUT` message is scheduled after the given time (lines 89–100).

Calls to `unlock`, `notify`, and `notifyAll` do not suspend a secondary thread. Instead, the operation is simply recorded in the action list of the corresponding entry in `PrimCandidates`, and later executed as soon as the action list is processed by `schedule`. A primary thread executes the three operations immediately. Unlock operations decrease the lock counter, and, if the counter reaches zero, remove the thread from `LockedMap` (lines 57–60). The notify operations (`notify` and `notifyAll`) move the first element or all elements, respectively, from `CondWaitMap` to `MutexWaitMap`, as the notified threads have to re-acquire the lock prior to continuation (lines 62–73).

If the primary thread issues a nested invocation, it calls `schedule` to select a new primary thread. If a secondary thread issues a nested invocation, this `schedule` call is delayed until it becomes primary. For this purpose, an action list entry is created. If `schedule` selects the thread as new primary, it processes the action list and re-calls `schedule`, because the current thread, which waits for a nested invocation reply, is not available as active primary thread. This means that a thread that waits for a nested invocation is never `ActivePrimary`, and it is neither in `MutexWaitMap` nor in `CondWaitMap`. As long as it is not referenced by `PrimCandidates` members, it is an NPC. The arrival of the nested invocation reply creates a `PrimCandidates` member with a reference to the thread, making it a primary candidate (PC).

Finally, the termination of a thread is noted in the action list of the `PrimCandidates` entry of the thread (line 49).

5 Verification of Our Algorithm

For verifying the correctness of the ADETS-MAT algorithm, we assume that all replicas have an identical initial state, no thread is initially active within the replicas and all synchronisation data structures are initialised with an empty state; the sequence of incoming messages is identical in all replicas, and the replica behaviour is piecewise deterministic.

The piecewise determinism of a replica implementation guarantees that, for an execution interval e_i of thread r (see Section 3), the local state L_r and the mutex-protected part of the shared state $S_{r,i}$ at the start of e_i uniquely defined the local and shared state at the end of e_i. While the local state only depends on message receptions (client request and nested-invocation replies, which are both consistently delivered to all replicas by total-order multicast) and on previous deterministic thread behaviour, the shared state is also influenced by the activity of other threads. The key problem in verifying ADETS-MAT thus is to show that these activities of other threads take place in a consistent order, i.e., that ADETS-MAT creates a deterministic schedule for mutexes.

In the following, we first show that each thread-execution interval has a deterministic effect on the scheduling data structures. This is specifically important if a thread-execution interval starts while a thread is not a primary; the executing thread can become primary at a nondeterministic point of time, either at any time during the thread-execution interval, or after the thread has suspend.

Lemma 1 (Deterministic Thread-Execution Interval) *Given a consistent local and shared state at the start of the execution interval e_i, the execution of e_i has a deterministic effect on the internal data structures of the scheduling algorithm.*

1) The piecewise determinism assumption guarantees that the behaviour of the *replica implementation* is deterministic during e_i.

2) If e_i starts by obtaining a mutex lock or by resuming from a `wait` operation, e_i will fully be executed by the primary thread. This means that all intercepted operations will call the same ADETS-MAT functions in all replicas, which will make deterministic modifications to the scheduler data structures.

3) If e_i is started by a client request or a nested-invocation reply, it is started as a secondary thread, creating a `PrimCandidates` entry. This entry can be processed by the scheduler at an arbitrary point in time, which means that some replicas can execute an intercepted operation as secondary, while others will execute the same intercepted operation as primary. It needs to be shown that both variants have the same final effect.

3.1) For `wait` and `lock` this is true, as a secondary thread issuing these operations simply blocks until it becomes primary.

3.2) For `unlock`, `notify`, and `notifyAll`, a secondary thread records the operations in the action list and then, after becoming active primary, executes the same steps as it would have made had it already been active primary.

3.3) On nested invocations and on thread termination, the active primary thread calls `schedule`, while the secondary instead adds an NESTED entry to the action list. After the secondary becomes primary, the NESTED entry causes the invocation of `schedule`, resulting again in a consistent behaviour.

In the following, we divide the progress within a replica into *rounds* R_i. Each round starts with the removal of an entry from `PrimCandidates` in `schedule` (line 12) and ends with the next invocation of `removeFirst` in the same line. After initialisation, `schedule` is called when the first element is added to `PrimCandidates`. `MutexWaitMap` is initially empty (line 3), and the invocation of `removeFirst` (line 12) returns the first element from `PrimCandidates`, starting the first round R_1. Subsequent rounds are numbered consecutively. A single round can consist of multiple *execution intervals*.

The following lemma shows that during each round, the ADETS-MAT algorithm behaves deterministically.

Lemma 2 (Consistent ADETS-MAT Behaviour) *During round R_i, the ADETS-MAT algorithm will make deterministic selections of the active primary thread and will make deterministic modifications to the scheduler data structures, given deterministic behaviour in all preceding rounds $R_k, k < i$.*

1) Initially, the head entry m from `PrimCandidates` is removed. By assumption, the sequence of received messages (and, consequently, of `PrimCandidates` entries) is identical in all replicas.

2) After removing m from `PrimCandidates`, the `schedule` function first processes the action list of m. If the action list contains a `TIMEOUT` message, it does not contain any other entries. The effect of such an entry m is to notify the mutex by deterministically moving it from `CondWaitMap` to `MutexWaitMap`, if the mutex is still waiting. After that, `schedule` is called.

3) Otherwise, the entry m references a real thread and its action list can contain an arbitrary sequence (zero or more elements) of UNLOCK and NOTIFY/NOTIFYALL entries, optionally followed by a TERMINATE, WAIT, LOCK, or NESTED entry. The referenced thread started with a shared state that only depends on previous rounds, which had a deterministic effect by assumption. By Lemma 1, the thread will cause a deterministic scheduler behaviour, independent of the time within its current execution interval at which it becomes active primary. At the end of the execution interval, `schedule` is called.

4) Subsequently, `schedule` iterates over the entries in `MutexWaitMap`, selecting a new active thread or, if no suitable thread is found, terminating the round. The selection of the new active thread is deterministically defined by the content of `MutexWaitMap` and `LockedMap`. By Lemma 1, the interactions of this thread with the ADETS-MAT algorithm will result in consistent modifications to the scheduling data structures in all replicas until the thread suspends or terminates, where it calls `schedule` to re-start the procedure of (4).

5) After the round has terminated, it directly follows from steps 1–4 that the same threads have been selected as active primary thread and that the scheduling data structures at the end of the round are deterministically defined.

Given a consistent initial state, by induction on i Lemma 2 implies that the scheduler activates the same threads and makes consistent modifications to its data structures for all rounds.

6 Evaluation

The algorithm proposed in this paper allows the concurrent execution of multiple threads within a replicated object. Our algorithm is superior to SAT-based algorithms, as the secondary threads add additional concurrency and thus permit true multithreading. To the best of our knowledge, the *Loose Synchronisation Algorithm* (LSA) and the *Preemptive Deterministic Scheduling* (PDS) algorithm are the only previously published strategies that similarly support concurrent request execution in object replicas [2–4]. In this section, we first compare our approach with PDS and LSA, and then provide an experimental evaluation.

6.1 Comparison with PDS

The PDS algorithm [2] divides the execution of concurrent threads into rounds. In each round, all threads may acquire one or two mutex locks; a set of complex rules is used to decide when it is safe to grant a lock to a thread. If threads request non-conflicting locks in a single round, they can execute in parallel. This can be superior to our algorithm, in which the next lock is only granted after a new active primary is selected. On the other hand, the PDS algorithm only starts a new round after all existing threads have suspended. As long as a single thread remains running, all others have to wait. The algorithm also assumes a fixed set of running threads. Client requests need to be assigned synchronously to these threads; the lack of new client requests can impede the start of a new round, unless the system generates pseudo-requests to avoid that problem. In contrast, in our algorithm any incoming client request can asynchronously create a new secondary thread. A further difference is that the PDS algorithm does not provide support for condition variables or nested invocations.

6.2 Comparison with LSA

The LSA algorithm [4] uses a leader-follower approach to provide deterministic scheduling. A single leader replica executes threads concurrently without any restriction and broadcasts the order of lock assignments to all other nodes. This approach achieves the best concurrency at the primary, as no scheduler constraints exist that can force a thread to wait for the acquisition of an available lock. The LSA algorithm, however, causes additional communication for synchronisation operations, and, in case of a primary failure, requires a complex reconfiguration procedure. In contrast, our approach, as well as PDS, operate fully locally on each node and do not require additional communication for synchronisation. The LSA achieves best efficiency if the leader broadcasts its synchronisation message asynchronously. In this case, however, a leader may return a result to a client and subsequently crash; the reconfiguration only guarantees the consistency of all surviving replicas, but not that they compute the same result as the original leader. This consistency problem is avoided if the leader broadcasts its lock order synchronously or if majority voting on replies is used. In these cases, however, the broadcast transmission time increases the invocation time that the client observes.

6.3 Experimental Evaluation

The following experiment is a simple representative pattern for an object in which methods first compute, for example, the verification of cryptographic certificates passed by the client and further preprocess the request arguments, and then update the object state protected by a mutex lock. A varying number of clients (1–10) invoke methods at an object replicated on 3 nodes. Each method locally computes for a time T randomly distributed from 0–20ms, then requests a mutex lock, modifies the object state, and unlocks the mutex. In the experiment, multi-CPU hosts for replicas are simulated by waiting locally instead of performing real computations.

The measurements were made on a set of AMD Opteron 2.2 GHz PCs running Linux 2.6.15 and connected via a switched 100 MBit/s Ethernet. We used the Aspectix ORB [8] (internally based on JacORB 2.2.1), JGroups 2.2.9.1 for group communication [1], and the Java Server VM 1.5 from Sun Microsystems. The JGroups stack was configured to use TCP connections and TOTAL message ordering. Each replica and client was placed on a separate host; clients iteratively issued remote invocations to the replica group.

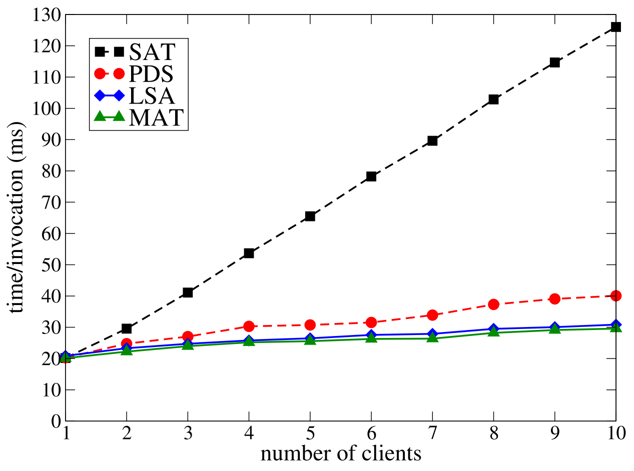

Figure 6. Average invocation times

Figure 6 shows the average time per remote method invocation observed at the client side. The SAT measurement uses a single-active-thread algorithm based on [14]. In this variant, the average time increases by about 10ms for each additional client; this corresponds to the average computation time of the additional request. Our ADETS-MAT algorithm allows computations to be executed in parallel on multiple CPUs, resulting in an almost constant invocation time independent of the number of clients (this behaviour implies that sufficiently many CPUs are available). An implementation of Basile's PDS algorithm performs better than SAT, but is less efficient than the ADETS-MAT algorithm. The main reason for this difference is that in PDS, the duration of each round is determined by the thread with the longest computation time. The LSA algorithm performs similar to our algorithm; the additional overhead due to the communication is slightly visible. In case of a primary failure, however, LSA would introduce a delay due to the reconfiguration, which does not happen with the ADETS-MAT algorithm.

6.4 Improvements for Further Increasing Concurrency

In the presented ADETS-MAT algorithm, the active primary thread can prevent all other threads from acquiring locks. If the active primary performs a computation of long duration, no other threads are allowed to acquire locks during this interval. This problem can be solved by (1) permitting multiple active threads and (2) turning the active primary thread into a secondary thread.

The active primary thread is responsible for determining a consistent order of synchronisation operations. Multiple primary threads can be used if they do not interfere with their synchronisation operations. For example, a thread t_1 that only operates on mutex m_1, and a thread t_2 that only operates on mutex m_2 can simultaneously be selected as primary threads without violating determinism. The property that threads do not interfere could be explicitly specified by developer annotations. To some extent, it could also be automatically be derived by automated code analysis.

The alternative is to turn the active primary thread into a secondary NPC thread when it is going to perform an extensive local computation. A replica implementation can actively requests such a transition by calling a `yield` method which selects a new active primary thread, making the thread that called `yield` a running secondary NPC thread. A group messages needs to be sent to the group to move this thread to `PrimCandidates` (and, ultimately, let it become active primary again). This way, long-duration computations can be used with our algorithm without inhibiting the lock acquisition of other threads.

7 Summary

In this paper, we have discussed strategies for handling multiple threads in replicated objects and have presented the novel ADETS-MAT algorithm for deterministic thread scheduling. The algorithm offers all benefits of a true multithreaded execution model: it avoids all potential deadlock problems of other execution models, and it allows multiple threads within an object to use all computational resources of multi-core CPUs or multi-CPU hosts. In addition, our approach is more flexible than previously published algorithms. It supports nested invocations, reentrant mutex locks, condition variables, and timeout-based interruption of wait operations on condition variables. Deterministic lock assignment is done completely locally on all replicas without communication. New threads may be created at any time to handle new client requests, and threads may access mutex locks in an arbitrary way.

References

[1] B. Ban. Design and implementation of a reliable group communication toolkit for Java. Technical report, Dept. of Computer Science, Cornell University, 1998.

[2] C. Basile, Z. Kalbarczyk, and R. Iyer. Preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proc. Int'l Conf. on Dependable Systems and Networks (DSN), 2003.*, 2003.

[3] C. Basile, Z. Kalbarczyk, and R. K. Iyer. Active replication of multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, 17(5):448–465, 2006.

[4] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. Iyer. Loose synchronization of multithreaded replicas. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, page 250, Washington, DC, USA, 2002. IEEE Computer Society.

[5] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[6] J. Fraga, C. Maziero, L. C. Lung, and O. G. L. Filho. Implementing replicated services in open systems using a reflective approach. In *ISADS '97: Proceedings of the 3rd International Symposium on Autonomous Decentralized Systems*, page 273, Washington, DC, USA, 1997. IEEE Computer Society.

[7] R. Friedman and A. Kama. Transparent fault tolerant Java virtual machine. In *SRDS '03: Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, pages 319–328, Washington, DC, USA, 2003. IEEE Computer Society.

[8] F. J. Hauck, R. Kapitza, H. P. Reiser, and A. I. Schmied. A flexible and extensible object middleware: CORBA and beyond. In *Proc. of the Fifth Int. Workshop on Software Engineering and Middleware*. ACM Digital Library, 2005.

[9] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *SRDS '00: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, page 164, Washington, DC, USA, 2000. IEEE Computer Society.

[10] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems — the Mars approach. *IEEE Micro*, 9(1):25–40, Feb. 1989.

[11] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant Java virtual machine. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003), DCC Symposium*, pages 425–434, June 2003.

[12] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *SRDS '99: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, page 263, Washington, DC, USA, 1999. IEEE Computer Society.

[13] S. Poledna, A. Burns, A. J. Wellings, and P. Barrett. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Trans. Computers*, 49(2):100–111, 2000.

[14] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Deterministic scheduling for multithreaded replicas. In *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 74–81, Washington, DC, USA, 2005. IEEE Computer Society.