Deterministic Scheduling for Multithreaded Replicas *

Wenbing Zhao
Department of Electrical and Computer Engineering
Cleveland State University, Cleveland, OH 44115
wenbing@ieee.org

L. E. Moser and P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
moser@ece.ucsb.edu, pmms@ece.ucsb.edu

Abstract

Replication of objects, processes and components is essential for building reliable distributed systems. However, maintaining replica consistency in the presence of nondeterminism is a challenge. There are many sources of nondeterminism in the applications being replicated. In this paper we focus on the nondeterminism resulting from multithreading in the applications, and present a deterministic scheduling algorithm for ensuring consistent execution of replicated multithreaded applications.

1 Introduction

In a distributed system, fault tolerance can be achieved by replicating each object, process and component of the applications on separate computers. If one computer becomes disabled by a fault, the replicas on the other computers are still available to execute the applications. It is essential that all of the replicas of a process, object or component perform the same operations, maintain the same state and thus are consistent so that, if one replica becomes faulty, the service provided by the applications is not disrupted.

Unfortunately, many applications are nondeterministic and, unless constrained, the replicas are liable to make different decisions and produce different results, causing their computations to diverge. The most challenging sources of nondeterminism are the interleaving of concurrent threads within a process and the order in which mutexes are granted to threads. To maintain replica consistency, the same interleaving of concurrent threads and the same order of granting mutexes must be maintained at all of the replicas. It is not appropriate to make those decisions at a single central point, because that central point might be a single point of failure.

*This research was partially supported by MURI/AFOSR Contract F49620-00-1-0330 at the University of California, Santa Barbara, and a faculty startup fund (for the first author) at Cleveland State University.

In deterministic scheduling, the fault tolerance infrastructure at each replica unilaterally makes decisions as to when to deliver a new request or reply to the replica. Ideally, it does so without sending extra control messages to coordinate the execution of the replicas and, yet, makes the same decision for all of the replicas and, hence, achieves replica consistency.

The two main replication strategies are active replication and passive replication.

In *active replication*, all of the replicas perform the requested service for the clients. Therefore, a fault in a single replica is masked with no interruption in the service. Active replication is appropriate if very fast recovery from faults is required.

In *passive replication*, one of the replicas is designated as the *primary* and the other replicas act as *backups*. Only the primary replica executes the requested service. The state of the primary replica is periodically checkpointed and the checkpoint is transferred, and applied, to the backup replicas. A backup replica logs incoming requests, so that it can take over as the primary replica when the primary replica fails.

Passive replication has been advocated if the potential for replica nondeterminism exists. However, the same replica nondeterminism problems that arise for active replication during normal operation arise for passive replication when the primary replica fails. Therefore, deterministic scheduling is important for ensuring strong replica consistency, regardless of the replication strategy being used.

The main challenge in developing a deterministic scheduling algorithm is to minimize the reduction in concurrency of the multithreaded application. Prior research [5] proposed an approach that forces a logical single thread of control, where requests are delivered and processed in sequential order. Such an approach preserves strong replica consistency. However, it can seriously degrade the performance of the applications when the applications use multiple disk inputs/outputs and nested invocations. Moreover, in some cases, a deterministic scheduling algorithm based

on a logical single thread of control might cause the application to deadlock, as we show in Section 3.

A deterministic scheduling algorithm for transactional multithreaded replicas is given in [3]. The algorithm uses two levels of input message queues: a common queue that stores the received totally ordered requests and replies, and local queues that store the requests retrieved for the threads from the common queue. A thread attempts to process the next message in its local queue only when every thread in the replica is blocked waiting either for a lock or for a message. A thread that attempts to obtain a new request from its local queue is blocked if the local queue is empty. When all of the threads are blocked or there are no running threads, the deterministic scheduler extracts a message from the common queue. If the extracted message is a request, it is appended to the local queue of the appropriate thread. If the message is a reply, a waiting thread is unblocked. The scheduling algorithm is implemented on top of a special-purpose proprietary library, called Transactional Drago [7]. The library fully controls the input message buffers and the creation, deletion and scheduling of threads.

The scheduling algorithm presented in [3] is more aggressive than that presented in [5], and it allows better concurrency. However, it is applicable only for applications that are programmed using the transaction processing paradigm with strict two-phase locking. Moreover, it requires full control of both message scheduling and thread scheduling.

More recently, a preemptive deterministic scheduling algorithm was proposed in [1]. However, that algorithm works only when all of the threads in a replica acquire the same number of locks for each request. The algorithm proceeds in rounds. In each round, the algorithm blocks the thread that attempts to acquire a lock until all threads have attempted to acquire a lock (not necessarily the same lock). At that point, both the set of locks and the set of threads that might compete for the locks are known and, thus, a deterministic decision can be made. Depending on the criteria being used, a thread that attempts to acquire a lock early might be preempted from acquiring the lock, which is why the algorithm is called preemptive.

The deterministic scheduling algorithm that we present in this paper is inspired in part by [3]. We aim for an algorithm that works with standard threading libraries and general-purpose multithreaded applications.

2 System Model

Modern distributed applications are usually built on top of commercial-off-the-shelf middleware, such as CORBA [6], that provides platform-independent high-level communication and concurrency control services. The application developers make decisions as to what kind of concurrency is appropriate for each object, based on policies, and those decisions are passed to the middleware, which executes accordingly.

Common concurrency models are the *thread-per-object*, *thread-per-client*, *thread-per-request*, *thread-pool*, and *thread-per-transaction* models.

In the *thread-per-object* model, all remote method invocations of the same object are handled by the same thread and, hence, they are naturally serialized. Invocations on different objects are handled concurrently by different threads.

In the *thread-per-client* model, invocations from the same client are handled by the same thread and requests from different clients are processed concurrently by different threads.

In the *thread-per-request* model, each incoming request is handled by a different thread. In the most naive implementation, a new thread is created to handle each incoming request and the thread is garbage-collected when the processing is done.

In the *thread-pool* model, a pool of threads is created during initialization of the application and the threads in the pool are used to handle incoming requests concurrently and efficiently. The thread-pool model can be used in conjunction with other concurrency models.

The *thread-per-transaction* model is used in transactional applications where all requests that belong to the same transaction are handled by the same thread. In transactional applications, it is essential to enable concurrent processing of different transactions. Both the algorithm given in [3] and our algorithm allow such concurrency, while the algorithm given in [5] might prevent multiple transactions from making progress concurrently. The algorithm given in [1] is not applicable to general transactional applications.

We assume that the replicated applications are implemented in such a way that the deterministic scheduling algorithm has access to the concurrency control policy for the objects. We also assume that all shared data in the application being replicated are protected by explicit locks (mutexes). The lock acquisition and release events are known to the fault tolerance infrastructure.

We further assume that the messaging protocol used by the middleware is known to the fault tolerance infrastructure so that, on receiving a request, it can retrieve information regarding the destination object and, subsequently, obtain the corresponding concurrency control policy. If the *thread-pool* model is used, additional information is required to associate an incoming request with the thread that processes the request.

We assume that the applications communicate *synchronously*, *i.e.,* if a thread issues a remote method invocation (sends a request message), then the thread blocks until it receives the corresponding reply. On receiving a request (from the scheduler), a thread dedicates itself to the

handling of the request until the corresponding reply is produced and sent to the lower layer. The thread subsequently attempts to obtain a new request. If no new request is available, or the message scheduler refrains from delivering one, the thread is blocked.

If all of the threads in a replica are blocked waiting for new requests, the replica is said to be *quiescent*. If all of the threads in a replica are blocked and the blocking of each thread is caused by either: (1) waiting for a request, (2) waiting for a reply, or (3) waiting for a lock (to access shared data), the replica is said to be *blocked*. The *blocked* state is less restrictive than quiescence, *i.e.,* if a replica is quiescent, it must be blocked; however, the converse is not true.

Starting from a quiescent state, the delivery of a new request activates a thread in the replica and the replica becomes unblocked. When the replica issues a nested request or a reply for a previous request, the replica becomes blocked.

Our deterministic scheduling algorithm operates on top of a reliable totally-ordered multicast group communication layer such as Totem [4] within a fault tolerance infrastructure such as PluggableFT [10]. The scheduler runs on a separate dedicated thread that the fault tolerance infrastructure creates and controls. The scheduler attempts to schedule a message for delivery periodically and also on events such as the receipt of a message from the network or the sending of a message to the network through the group communication layer.

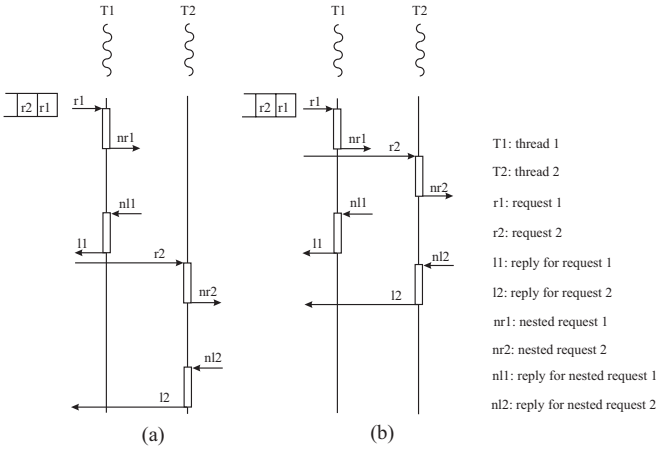

Figure 1. A comparison of two deterministic scheduling algorithms: (a) logical single thread of control algorithm, and (b) our deterministic scheduling algorithm.

3 Deterministic Scheduling Algorithm

3.1 Intuition and Reasoning

According to the logical single thread of control scheduling strategy of [5], a new request is delivered to a replica only when the replica has issued a reply for the previous request. No request message is delivered to the replica if the replica is blocked waiting for the reply to a nested request, as shown in Figure 1(a). However, it is actually safe to deliver the next request to the replica as soon as the replica is blocked (because no thread is active at that moment), *e.g.,* immediately after the replica issues a nested invocation, as shown in Figure 1(b). The delivery of a new request can unblock another thread. Therefore, the replica can continue processing in a new thread while the previous thread is blocked waiting. Both the algorithm given in [3] and our algorithm are based on this observation.

The logical single thread of control strategy of [5] can have more negative consequences than degradation in performance due to decreased concurrency, such as introducing deadlocks in a system that does not suffer from deadlocks otherwise. Figure 2 illustrates such a deadlock scenario for a simple distributed banking application. Client 1 issues a *commit()* remote method invocation of the transaction coordinator to commit a distributed transaction. At the same time, client 2 issues a *deposit()* remote method invocation of the server application (*i.e.,* one of the transaction participants) as part of the transaction. During the processing of the *commit()* request, the transaction coordinator issues a *prepare()* remote method invocation to the server application. At the same time, the server application issues an *_is_a()* remote method invocation of the transaction coordinator to confirm the object reference for the *control* object for the transaction that client 2 started.

With the logical single thread of control strategy, the fault tolerance infrastructure does not deliver a new request to the replica until the previous request has been fully processed and the reply from the replica has been received. This constraint prevents the infrastructure at the transaction coordinator or the server application from delivering the nested invocation (*prepare()* or *_is_a()*). However, neither the transaction coordinator nor the server application can become quiescent without receiving the reply for the nested invocation, which will never arrive because the replica is not quiescent. Therefore, the system reaches a deadlock.

The task of a deterministic scheduler is complicated, because (1) a replica might receive multiple messages that the same thread must handle, and (2) a thread might acquire a mutex and not release it until a number of nested invocations are completed (which is likely for transactional processes).

In case (1) it is not safe to deliver the next request message to a thread if the thread has not fully processed the previous request message, even if it is blocked (*e.g.*, the thread might be waiting for a reply to a nested request that it is-

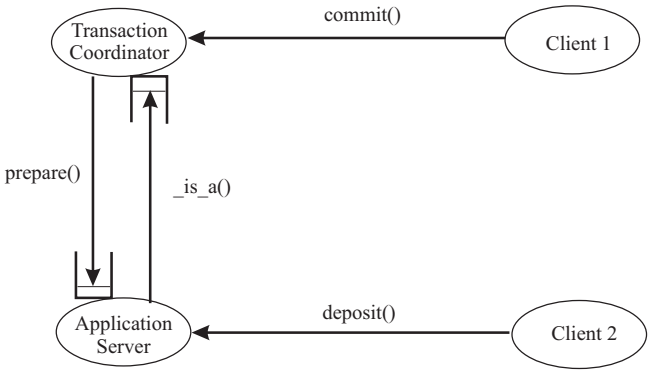

Figure 2. Deadlock under the logical single thread of control scheduling algorithm for a simple distributed banking application.

sued earlier, or it might be waiting for a mutex). The reason is that a large queue of messages might form for the thread and, if selective reception [3] starts, different replicas might not see the same set of messages (*i.e.*, even though messages are totally ordered, a message might be delivered to one replica before the delivery of the message to another replica in real time). Furthermore, if a queue is formed for each replica, it is difficult for the scheduler to determine if the replica is in a blocked state (because the threading library is treated as a black box).

To cope with case (1), the scheduler does not deliver a message to a replica if the delivery of the message does not unblock a thread, even if the replica is in a blocked state. Instead, it examines the next message in the message queue to see if it can unblock a thread. In general, a new request is delivered only when the thread is quiescent, *i.e.,* the thread has processed all requests delivered to it and is not in a critical section.

In case (2), it might not be safe to deliver a new request message because the replica might have multiple threads running concurrently and their interleaving is not controlled by the fault tolerance infrastructure.

Consider the scenario shown in Figure 3. Thread T1 acquires a lock L1 while it processes request message r1. Before it releases lock L1, T1 issues a nested request nr1, at which point the thread and the replica as a whole are blocked. An incoming request message r2 is delivered to a different thread T2. However, T2 also wants to acquire L1 in order to access the shared data protected by L1. Therefore, thread T2 is blocked. A short while later, the reply nl1 for the nested request nr1 issued by thread T1 arrives and is delivered to T1. While it processes the reply nl1, thread T1 releases lock L1 and continues processing. The release of lock L1 unblocks thread T2.

Starting from this point, there are two threads (T1 and T2) running concurrently. If the two threads do not access the same shared data from then on, no inconsistency will result from the concurrency. However, this is not guaranteed.

As shown in Figure 3, if threads T1 and T2 later try to access the shared data protected by lock L2, T1 might acquire L2 ahead of T2 at one replica (as shown in Figure 3(a)) while T2 might acquire L2 ahead of T1 at another replica (as shown in Figure 3(b)). This behavior can lead to inconsistency in the states of the two replicas because the shared data protected by L2 might have different values as a result of different accesses by the two threads T1 and T2.

Other scenarios might also arise due to the delivery of new request messages. For example, suppose that one thread in a replica has acquired a number of mutexes and then issues a nested remote method invocation. Subsequently, a new request message is delivered to the replica. The thread that processes the new message would be blocked if it needs to acquire a mutex that is held by the first thread. Potentially more threads would be blocked if they also need to obtain the mutexes held by the first thread on delivery of new messages. When the reply message to the nested request issued by the first thread is delivered to the replica, the first thread might release multiple mutexes that it acquired earlier and that might activate multiple blocked threads. If some of these threads later compete for the same mutex, the order of the mutex acquisition is not controlled by the scheduler.

As the above examples show, without application-specific knowledge or additional mechanism, the scheduler cannot be made more aggressive to ensure strong replica consistency.

In our initial study of this problem, we reverted to the logical single thread of control strategy for case (2), *i.e.,* no new message is delivered to any other thread until the message is received by the thread in the critical section. Unfortunately, such a strategy can prevent concurrent processing.

The algorithm in [3] resolves this problem by maintaining a ready-thread queue and activating only the first thread in the ready-thread queue when the existing active thread returns control to the scheduler. However, that approach is not directly applicable to our algorithm because our scheduler is external to the threading library and, hence, cannot directly control the internal thread scheduling (for example, if the active thread reaches a scheduling point outside our control, a ready thread will be activated subsequently without our knowledge).

The cause of the inconsistent execution of different replicas is the running of multiple threads concurrently. However, if the scheduler ensures that only one thread runs at a time and that the ordering of the execution of threads is the same at different replicas, replica consistency can be achieved by control over the unblocking of threads that are

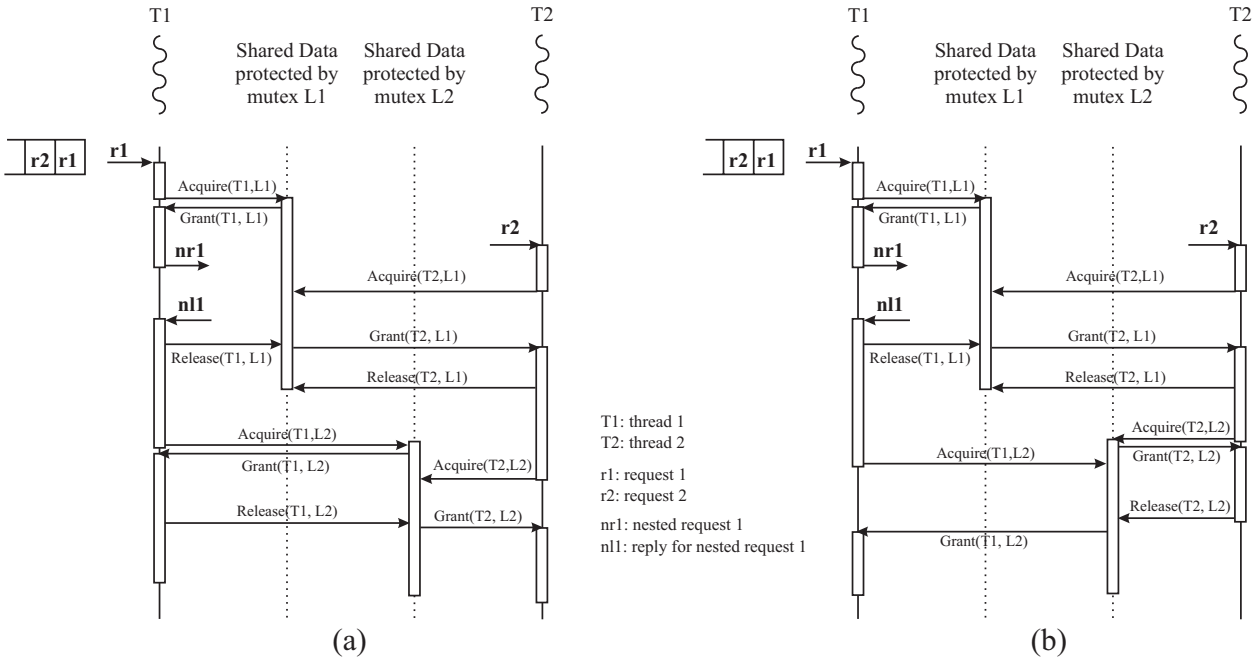

Figure 3. If a thread in a replica still holds a lock while the replica is blocked, the delivery of an incoming message to another thread can result in inconsistency between two replicas. (a) In one replica, thread T1 updates the shared data protected by the second lock before thread T2 does so. (b) In another replica, thread T2 updates the shared data protected by the second lock before thread T1 does so.

waiting for mutexes after the mutexes are released. This strategy involves two steps. First, instead of unblocking a waiting thread as soon as a mutex is released, the waiting thread is unblocked only when the thread that previously held the mutex is blocked on a message or another mutex. Second, if there are two or more threads that would be unblocked by releasing the mutex, the scheduler activates only one such thread at a time. Thus, the problem of uncontrollable concurrent execution of multiple threads is eliminated. Furthermore, the choice of the thread to unblock is deterministic, which ensures that the threads execute in the same order at all of the replicas.

3.2 Data Structures

For each replica, our deterministic scheduling algorithm employs the following local data structures:

- *thr_id*: Thread identifier.
- *mutex_id*: Mutex (lock) identifier.
- *msg_id*: Message identifier, consisting of the connection identifier, the message sequence number, and the role of the message (*i.e.,* request or reply). The reply for a request message is matched by inspecting the information in the message identifier.
- *msg_q*: The incoming message queue. Incoming messages arrive in this queue in a total order. However, the queue is a priority queue because a later incoming message (*e.g.,* a reply) might be delivered out of order. A message is taken out of the queue if the scheduler decides to deliver it and remains in the queue otherwise.
- *thr_table*: The thread table, which maintains the information regarding the threads that the replica has created. There is one record for each thread. Each record contains the following four fields:
  - *thr_id*: The thread identifier.
  - *acquired_mutex_ids*: The identifiers of the mutexes that the thread has acquired but has not yet released while processing a message delivered to the thread.
  - *outstanding_msg_dlvred*: The identifier of the message that has been delivered to the thread and for which the processing has not been completed.

– *outstanding_request_sent*: The identifier of the nested request message that has been sent by the thread and for which the corresponding reply is pending.

The thread identifier field is initially set to *nil* when a new record is created, because the middleware supporting the application determines the thread identifier on delivery of a new request. This field is populated when the new thread is activated by a new request. Similarly, the other three fields are initially set to *nil*.

- *blocked_thr_q*: The blocked thread queue is a priority queue that stores the identifiers of the threads that are blocked on locks to access shared data. When a thread attempts to acquire a mutex that is held by another thread, or the mutex cannot be granted to the calling thread for deterministic scheduling reasons, the thread identifier is appended to the queue together with the mutex identifier. Note that a thread that is blocked on an incoming message is not recorded in this queue; rather, such information is stored in the thread table.

- *conc_ctrl_mgr*: The concurrency control manager object is the entity by which the scheduler has access to the concurrency control policy for the object to which a request message is delivered.

The incoming message queue and the thread table both have public methods for the scheduler to access and manipulate their state. The incoming message queue has the following methods:

- *begin()*: This method returns an iterator object to the caller. The caller can iterate through the messages in the message queue by using the iterator object.

- *end()*: This method returns a *nil* iterator to the caller, so that the caller knows when the end of the message queue is reached.

- *remove_msg()*: This method takes an iterator object as an input parameter and removes the corresponding message from the message queue.

The thread table has the following methods:

- *add_msg_dlvred()*: This method adds the identifier of the message, delivered to the replica, to the thread table. The identifier of the thread that is to process the message, if known, is also passed as a parameter, in order to locate the corresponding record in the table. If a new thread is to be created, a *nil* value is passed in and a new record is inserted into the table. When the creation of a new thread is detected, the identifier of the new thread is added to the table. Note that the deterministic scheduling algorithm allows the creation of only one thread at a time, so there is no ambiguity in associating the newly created thread with the corresponding record in the table. If a reply is delivered to a thread, the thread must have issued a nested invocation and the corresponding nested request field in the thread table is cleared.

- *add_msg_sent()*: This method adds the identifier of the message, sent by the replica, to the thread table. The sending thread identifier is passed in as a key to match the particular record in the thread table. If a reply is sent by a thread, the corresponding request message field is cleared because, at this point, the request has been fully processed. Moreover, the *released_mutex_ids* field is cleared.

- *acquire_mutex()*: When a thread is to acquire a mutex, this method is invoked to update the record that corresponds to the thread. If the mutex is not currently held by any other thread and the scheduler permits, the mutex is granted to the calling thread. Otherwise, the calling thread is blocked. If the calling thread is blocked, the *released_mutex_ids* field in the thread record is cleared.

- *release_mutex()*: When a thread is to release a mutex, this method is invoked to transfer the identifier of the mutex from the *acquired_mutex_ids* field to the *released_mutex_ids* field.

- *block_thread()*: This method takes the identifier of the calling thread and the identifier of the mutex that the thread wants to acquire. When it is invoked, a search is carried out to see if another thread currently holds the mutex, or if the thread that released the mutex recently is still processing, in which case the mutex acquisition attempt is normally blocked. If a record can be found that contains either the mutex identifier in the *acquired_mutex_ids* field or the *released_mutex_ids* field, the calling thread is normally blocked. The only exception is that, if the calling thread is the thread that previously acquired and released the mutex while processing the current message delivered to it, the thread is granted the mutex.

- *unblock_thread()*: This method takes a thread identifier and a message identifier as input parameters and returns true if the delivery of the message unblocks the thread. The decision to unblock the thread is made based on the record for the thread. If the thread is blocked waiting for a new request, the delivery of a new request would unblock the thread. If the thread is blocked waiting for a reply (as a result of sending a nested invocation), the delivery of the reply unblocks the thread.

- *is_blocked()*: The scheduler invokes this method to see if the replica is in the blocked state, *i.e.,* all threads in the replica are blocked on a mutex or a message.

The blocked thread queue has the following methods:

- *enqueue()*: This method appends the calling thread identifier and the associated mutex identifier to the blocked thread queue when a thread attempts to acquire a mutex, but the mutex is held by another thread or the mutex cannot be granted for deterministic scheduling reasons.

- *unblock_thread()*: When this method is invoked, the entries in the blocked thread queue are examined starting from the oldest. The thread table is then consulted to see if the scheduler allows the unblocking of the calling thread. If so, the entry is removed from the queue and the mutex is granted to the calling thread. The thread then becomes active. Note that this method is invoked only when the replica has reached a blocked state. Even if two or more threads in the queue can be unblocked, only one thread is activated at a time according to first-in first-out order.

The *conc_ctrl_mgr* object has the following method:

- *get_proc_thread()*. This method takes a message as an input parameter and returns the identifier of the thread that is to handle the message. If a new thread is to be created, it returns a *nil* value.

```
   On interception of a mutex claim (mutex_id):
   begin
1      thr_id ←— calling thread id;
2      if thr_table.would_block_thread(thr_id, mutex_id) = true then
3          blocked_thr_q.enque(thr_id, mutex_id);
4      thr_table.acquire_mutex(thr_id, mutex_id);
   end
   On interception of a mutex release (mutex_id):
   begin
5      thr_id ←— calling thread id;
6      thr_table.release_mutex(thr_id, mutex_id);
   end
   On sending a message (msg_id):
   begin
7      thr_id ←— sending thread id;
8      thr_table.add_msg_sent(thr_id, msg_id);
   end
   On selecting a message to deliver:
   begin
9      if thr_table.is_blocked() ≠ true then
10         return nil;
       // See if a thread that blocks on a mutex can now be unblocked;
11     if blocked_thr_q.unblock_thread() = true then
12         return nil;
13     Iterator i ←— msg_q.begin();
14     while i ≠ msg_q.end() do
15         Message msg_dlvr ←— i.get_msg();
16         Thread_id thr_id ←—
               conc_ctrl_mgr.get_proc_thread(msg_dlvr);
17         if thr_table.would_unblock_thread(thr_id) = true then
18             Msg_id msg_id ←— msg_dlvr.get_id();
19             thr_table.add_msg_dlvred(thr_id, msg_id);
20             msg_q.remove_msg(i);
21             return msg_dlvr;
22         i++;
23     return nil;
   end
```

Figure 4. Pseudocode for the deterministic scheduling algorithm.

3.3 Deterministic Scheduling Algorithm

The pseudocode for our deterministic scheduling algorithm, which is based on the above observations and data structures, is given in Figure 4. When an application thread attempts to acquire a mutex, the thread table is first consulted to see if the mutex acquisition is allowed. If not, an entry is appended to the blocked thread queue (lines 1-3). Subsequently, the call is dispatched to the thread table (line 4). If the mutex acquisition is permitted, the corresponding entry in the thread table is updated and control is returned to the calling thread. Otherwise, the calling thread is blocked waiting on the mutex.

When a thread wants to release a mutex, the entry in the thread table is updated (lines 5-6). Internal to the *release_mutex()* method, the mutex identifier is removed from the *acquired_mutex_ids* field and is added to the *released_mutex_ids* field. Control is then returned to the calling thread. However, the mutex is implicitly associated with the previous owner (as if it still holds the mutex) and no other thread that is waiting for the mutex is unblocked until the previous thread holding the lock reaches a blocked state.

When a thread sends a message (issues a remote method invocation), the record in the table that corresponds to the sending thread is updated (lines 7-8). If the message sent is a request message, the *outstanding_request_sent* field is set with the identifier of the (nested) request message. If the message sent is a reply message, the thread must have received a corresponding request message, *i.e.,* the *outstanding_msg_dlvred* must have been set to the request message and, therefore, the *outstanding_msg_dlvred* is set to *nil*. These operations are carried out within the *add_msg_sent()* method.

When the scheduler attempts to select a message for delivery, it first consults the thread table to see if the replica is in a blocked state. No message is scheduled if the replica is not in a blocked state (lines 9-10).

If the replica is in a blocked state, the scheduler first checks the blocked thread queue to see if a thread that is waiting for a mutex can now be unblocked (lines 11-12). If so, no message is selected because the replica can still make progress using the messages that were delivered previously.

If the unblocked thread queue is empty or no thread can be unblocked, the scheduler examines the messages in the incoming message queue starting with the oldest message (lines 13-23). The scheduler first obtains the identifier of the thread that is to process the message if delivered (lines

15-16). The message queue might have stored such information if the message has been examined previously, in which case the information is retrieved directly from the message queue (not shown in the pseudocode). Then, the scheduler consults with the thread table to see if the delivery of the message would unblock the thread (line 17). If so, the thread table is updated, the message is removed from the message queue, and the message is delivered to the application (lines 18-21). Otherwise, the scheduler keeps the message in the message queue and examines the next message in the queue (line 22). The same check is then carried out for the next message. It is possible that, after all messages in the message queue have been examined, no message can be delivered (line 23).

4 Conclusion

We have presented a deterministic scheduling algorithm for multithreaded replicas. The scheduling algorithm is conservative in that only one thread is run at a time to ensure consistent execution of the multithreaded replicas. The algorithm achieves better concurrency than the algorithm in [5] by delivering an incoming request as soon as possible and by employing knowledge of the concurrency control policy used by the replicas. In addition, the algorithm avoids the deadlock problems of the algorithm given in [5].

A higher degree of concurrency might be achieved by integrating application-specific knowledge into the deterministic scheduling algorithm as in [8]; however, that would result in loss of transparency to the application. Alternatively, a higher degree of concurrency might be achieved by resorting to explicit coordination among the replicas to ensure the same order of access to shared data at all replicas as in [2], but that would result in a negative performance impact.

References

[1] C. Basile, K. Whisnant, and R. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 149–158, San Francisco, CA, June 2003.

[2] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. Iyer. Loose synchronization of multithreaded replicas. In *Proceedings of the IEEE International Symposium on Reliable Distributed Systems*, pages 250–255, Suita, Japan, October 2002.

[3] R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. Deterministic scheduling for transactional multithreaded replicas. In *Proceedings of the IEEE 19th Symposium on Reliable Distributed Systems*, pages 164–173, Nurnberg, Germany, October 2000.

[4] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[5] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *Proceedings of the IEEE 18th Symposium on Reliable Distributed Systems*, pages 263–273, Lausanne, Switzerland, October 1999.

[6] Object Management Group. The Common Object Request Broker: Architecture and specification, 2.4 edition. OMG Technical Committee Document formal/2001-02-33, February 2001.

[7] M. Patino-Martinez, R. Jimenez-Peris, and S. Arevalo. Synchronizing group transactions with rendezvous in a distributed Ada environment. In *Proceedings of the ACM Symposium on Applied Computing*, pages 2–9, Atlanta, GA, February 1998.

[8] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 315–329, Toledo, Spain, October 2000, Lecture Notes in Computer Science, vol. 1914, Springer-Verlag, Berlin, Germany.

[9] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[10] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Design and implementation of a pluggable fault tolerant CORBA infrastructure. *Cluster Computing: The Journal of Networks, Software Tools and Applications, Special Issue on Dependable Distributed Systems*, 7(4):317–330, October 2004.