Deterministic Scheduling for Transactional Multithreaded Replicas *

Ricardo Jiménez-Peris†
Marta Patiño-Martínez†
Technical University of Madrid
Facultad de Informática
E-28660 Boadilla del Monte (Madrid), Spain
{rjimenez, mpatino}@fi.upm.es

Sergio Arévalo‡
Universidad Rey Juan Carlos
Escuela de Ciencias Experimentales
E-28933 Móstoles (Madrid), Spain
sarevalo@escet.urjc.es

Abstract

One way to implement a fault-tolerant a service is by replicating it at sites that fail independently. One of the replication techniques is active replication where each request is executed by all the replicas. Thus, the effects of failures can be completely masked resulting in an increase of the service availability. In order to preserve consistency among replicas, replicas must exhibit a deterministic behavior, what has been traditionally achieved by restricting replicas to be single-threaded. However, this approach cannot be applied in some setups, like transactional systems, where it is not admissible to process transactions sequentially. In this paper, we present a deterministic scheduling algorithm for multithreaded replicas in a transactional framework. To ensure replica determinism requests to replicated servers are submitted by means of reliable and totally ordered multicast. Internally, a deterministic scheduler ensures that all threads are scheduled in the same way at all replicas what guarantees replica consistency

1. Introduction

One way to implement a fault-tolerant service is by replicating it at several sites that fail independently. If active replication [18] is used all the replicas perform the requested services. Hence, failures are masked to the client as far as there is an operational replica.

In order to ensure replica consistency, all replicas must process the same requests in the same order and must behave deterministically. That is, replicas must behave as state machines [18]. The first of these requirements can be achieved using group communication primitives [10], in particular reliable total order multicast. This kind of multicast guarantees that all messages are delivered in the same order to all the available replicas. Deterministic behavior has been traditionally achieved by means of single-threaded replicas. However, the approach of single-threaded replicas might be too restrictive in some environments. For instance, in a CORBA server [14] or a transactional server, that approach is not applicable.

Although, there are some algorithms that allow multithreaded replicas [14], their use in a transactional context is not feasible, as only one request is processed at a time. In a transactional context, if each service is executed as a transaction and concurrency control is based on a pessimistic method (locks), a server would stop processing when a transaction is blocked on a data item. Therefore, incoming requests are not processed until that transaction finishes, even if they do not access the same data items.

In this paper we present a non-preemptive scheduling algorithm for multithreaded replicas in a transactional context. The algorithm ensures the deterministic scheduling of active replicas and it is able to execute several transactions concurrently. This algorithm is used in *Transactional Drago* [16], a distributed programming language that provides transactional replicated servers.

* This work has been partially funded by the Spanish Research Council *(CICYT)*, contract number *TIC98-1032-C03-01* and the Madrid Regional Research Council *(CAM)*, contract number *CAM-07T/0012/1998*.

This paper is organized as follows. First, the system model is described in Section 2. Then, the different sources of non-determinism are identified in Section 3. Section 4 presents a deterministic scheduling algorithm for multithreaded replicas (MTRDS). The correctness of the MTRDS algorithm is proven in Section 5. Section 6 presents some implementation issues to support the algorithm. Finally, we compare our approach to other works and present our conclusions.

2. System model

The system consists of a set of nodes interconnected by means of a network. We assume the nodes to be fail-silent and that there are no network partitions. We do not consider neither malicious failures (i.e., byzantine failures), nor the non-determinism introduced by software interrupts.

2.1. Communication model

A server provides a set of services that clients invoke by submitting requests. Servers are replicated, that is, each one consists of a group of identical replicas (i.e., with the same code and data). In order to increase the fault tolerance, replicas run at different sites. Clients communicate with replicated servers (from now on servers) using group communication primitives (multicast) [4]. These primitives can be classified attending to the order guarantees and fault-tolerance provided [10]. *Total order* ensures that messages are delivered in the same order to all the replicas. With regard to fault-tolerance, *reliable multicast* ensures that a message is delivered to all or none of the available replicas. Messages will be sent by means of reliable total order multicast. Client/server interaction is synchronous, i.e., the client remains blocked until it gets the reply.

2.2. Transaction model

Clients interact with servers within a transaction. A transaction can finish successfully (*commits*) or not (*aborts*, the effect is as if the transaction had not been executed). The outcome of a transaction is notified to servers by the underlying transaction processing system.

Transactions are partially ordered sets of read and write operations [3]. Two transactions conflict, if they access the same data item and at least one of them is a write operation. Locks are used for concurrency control purposes. They are requested before accessing a data item and released when the transaction finishes. Hence, if a transaction gets a write lock on a data item, other transactions accessing that item would be blocked until that transaction finishes. Once locks are released, they are granted in fifo order (i.e., there is a queue of blocked transactions).

A history H of committed transactions is serial if it totally orders all the transactions [3].

For replicated data, the correctness criterion is one-copy-serializability [3]. Using this criterion, each copy must appear as a single logical copy and the execution of concurrent transactions must be equivalent to a serial execution over all the physical copies.

Client transactions can be multithreaded. The client can spawn threads within a transaction, and all the threads are part of the same transaction.

2.3. Interaction with servers

The interaction with servers is conversational [7]. That is, a client can issue service requests that refer to earlier requests. The server knows what the client is allowed to do at a point of the interaction, and which results have been produced so far. One of the advantages of this kind of interaction is that the programming of a server is simplified as its code only deals with a single client. Furthermore, a server can easily impose a protocol of calls to its clients. For instance, a mail server creates the mail header during the first interaction with a client, the mail body during the next interaction, and so on. The mail server ensures that no mail is sent without a header and a body. Another advantage of this kind of interaction is that the server can perform processing without blocking the client (i.e., between calls).

A new thread is created at each replica for each different client (i.e., each client transaction). A thread executes an instance of the replica code and only processes requests from that transaction. Threads are created when the first call from a transaction is processed at a server. All threads of a replica share the data of the replica. The thread code itself must be deterministic.

Figure 1 shows this behavior with a two-replica server and two clients (`T1`, `T2`). At each replica there are two threads, one per client. Both threads have the same code and only process requests (`e1`, `e2`) from their transactions.

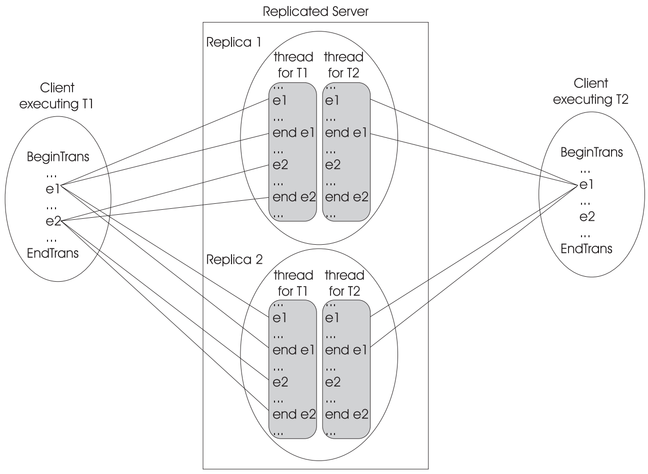

Figure 1. Interaction with a replicated server

A server can process a particular service or choose one among a set of services (selective reception). The example in Fig. 1 belongs to the former case. Replicas first process service e1, and then e2. The latter case would be a server that is willing to process either e1 or e2. If there are no requests for any of the services, the thread blocks until an awaited request is received. If there are requests for both services, one of them is selected and processed.

A replicated server can issue calls to other servers (that is, it can also act as a client). As all the replicas perform the same actions, a particular call is then issued by all the replicas. These calls should be filtered to avoid performing the call multiple times. This filtering guarantees that the call is executed exactly once. The reply is sent back to all the replicas. This filtering can be automatically done using a communication library providing *m to n* group communication [13], like GroupIO [8], or implementing this facility on top of a group communication library [14].

3. Enforcing determinism

In order to guarantee replica consistency determinism must be enforced.

The sources of non-determinism can be classified as external and internal.

The external environment of the replicas is a source of non-determinism. This external environment consists of all the messages the replica receives (client requests or transaction management messages).

Client requests could reach each replica in a different order. Then, they can be executed in a different order at each replica violating replica consistency. Executing conflicting operations in the same order at all replicas removes this source of non-determinism. In general, there is no way to know a priori whether two requests will have conflicting operations, thus all operations will be executed in the same order at all replicas. This can be achieved using reliable totally ordered multicast.

As a server can be client of other servers, replies to requests issued from a replicated server must also be received in the same order at all the replicas. But requests and replies are not the unique external events that replicas receive. Servers also receive transaction management messages. These messages must also be taken into account to achieve determinism. In particular, transaction termination (abort or commit) messages can release locks that will unblock threads (transactions) blocked on those locks. Therefore, transaction management messages must also be totally ordered multicast to guarantee replica consistency.

Providing replicas with the same external environment is not enough to guarantee the determinism of multithreaded replicas. Multithreading itself is an internal source of non-determinism. Two replicas A and B receiving two conflicting requests r_1 and r_2 in the same order can schedule the associated threads in different order, which will produce inconsistencies among the replicas. For this reason, it is necessary to provide replicas with a deterministic scheduler, which ensures that all replicas will perform the same thread scheduling. Given the same ordered set of messages and initial state, a *deterministic scheduler* produces exactly the same thread interleaving, provided that the code run by server threads is deterministic (i.e., it cannot use local services that yield different results from replica to replica, like the current time).

Unfortunately, total ordered requests and deterministic scheduling do not suffice to ensure replica consistency. Despite delivering messages in the same total order at all replicas, it is not guaranteed that all the replicas will deliver messages at the same time. That is, one replica can deliver messages faster than other replicas, what does not violate the total ordering. At a given scheduling point, a replica may have some queued messages (Fig. 2), while another one has no messages. The replica with messages could decide to process the first pending message, whilst the one with no messages can only decide to schedule one of its ready threads. That situation will compromise again the consistency of the replicas.

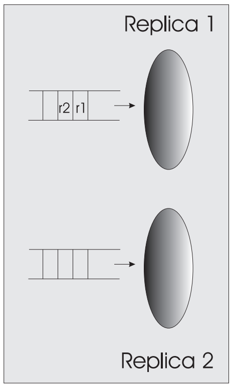

Figure 2. Unsynchronized queue states

To prevent this situation, a replica will process a new message when it is the only way to progress (i.e., all the threads of the replica are blocked or there are no running threads). Therefore, when a replica has to choose between processing a new message or scheduling a ready thread, it will always do the latter, removing the non-determinism. Another deterministic possibility is alternating the actions of processing a new message and scheduling a ready thread. This approach has a drawback. If a replica has no queued messages, it will block, while it could be processing a ready thread.

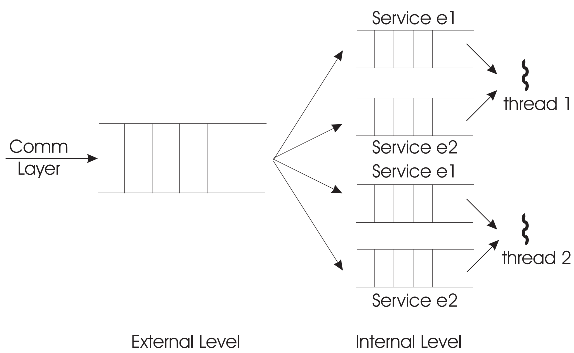

Figure 3. Message queue levels at a replica

This solution uses implicitly two message queue levels. The *external level* queue corresponds to the communication layer. There is one of such queues at each replica. The *internal level* queues correspond to the services provided by the server. Each replica thread has an internal queue per service. Hence, a message in one of internal queues will be a request from the associated transaction for that service. In (Fig. 3) these two levels are shown for a replica with two services, `e1` and `e2`, and running two server threads. Hence, each thread has two queues, one for each service. As far as it is possible to progress with the messages in the second level, no message is taken from the external queue. Messages taken from the external queue are moved to the corresponding second level queue. This process is repeated until a thread becomes ready.

Selective reception introduces a new source of non-determinism. As clients can be multithreaded, they can issue concurrent requests to a server. Therefore, each server thread can have queued an arbitrary number of pending requests. When a selective reception is reached, each server thread of a transaction can choose a different message among the ones that can be processed, introducing again replica inconsistencies. Thereby, in a selective reception messages must be chosen deterministically. Guaranteeing that every server thread always considers the same set of messages, and that the threads choose deterministically from this set of messages, it is enough to enforce determinism of selective reception. The two-level queue scheme ensures that all server threads will always consider the same set of messages (the ones in the second level). Therefore, it also solves the problem of selective reception. With this scheme the selection can even be performed randomly, as far as it is done in the same way at all replicas. This can be achieved by using a random number generator initialized with the same seed at all replicas.

4. The deterministic scheduling algorithm

The description of the *Multithreaded Deterministic Scheduling* algorithm (MTRDS) will be done in two steps. First, we introduce the data structures used by the algorithm (Section 4.1). Then, we describe how the algorithm works with the help of some examples (Section 4.2).

4.1. Algorithm data structures

The algorithm uses several data structures to keep its state. These structures are message queues, thread queues, and a thread table.

As it was discussed in the previous section, there are two levels of queues. In the external level (communication layer) messages are totally ordered. This queue holds client requests, replies, and transaction management messages. The internal level (server services queues) only holds client requests. Extracting messages from any of these queues is a potentially blocking operation.

That is, a thread trying to extract from an empty queue will be blocked until a message is queued.

There are queues for ready threads and for blocked ones. The queue of ready threads, as its name indicates, stores the threads that are ready to execute. The fifo extracting policy of the queue guarantees a fair scheduling among the ready threads. Each data item of a replica has a queue of blocked threads waiting for a conflicting lock. If the thread on execution, *active thread*, requests a conflicting lock on an item (i.e., an item already locked in a conflicting mode by a different transaction), the thread will be blocked and stored in the associated lock queue. Upon transaction termination, threads that are unblocked by the release of locks (if any) are moved to the queue of ready threads.

Finally, a thread table is used to keep track of the relation among clients and the server threads at a replica. When a request corresponds to a transaction that is not in the table, it means that the request is the first one from that transaction. Then, a new thread is created at that replica. Thread termination is also annotated in this table. This is useful to detect the case when a client calls the server once the interaction protocol has finished (a protocol violation).

4.2. The MTRDS algorithm

We will use an object-oriented notation to describe the algorithm. The objects involved in the algorithm are: messages (msg), the queue of ready threads (readyThreadQueue), the thread table (threadTable), threads (activeThread, serverThread) that also encapsulate their internal message queues, the transaction manager (transManager, for the sake of simplicity the lock, recovery and transaction managers are represented with this single object), system (an object providing thread management operations), and the external message queue (messageQueue). The methods for the different objects are summarized in Table 1.

In our approach a replica consists of a single process, whose main thread is the scheduler. The scheduler implements the MTRDS algorithm (Fig. 4) that is non-preemptive. The scheduler is in charge of creating new server threads. At a given time in a replica either the scheduler or a server thread is executing. The scheduler after performing a *scheduling step* transfers the control to a ready thread (that becomes the *activeThread*). The active thread returns control to the scheduler when it reaches a *scheduling point* (due to the non-preemptive nature of the scheduler). The scheduling points are acceptance of a service request, selective reception, lock request, server call, and end of execution.

If the active thread returns control to the scheduler without blocking (accepting a message from a non-empty service queue or accessing a non-conflicting data item), it is stored in the queue of ready threads. Otherwise, it will be already blocked awaiting a message or a lock. Then, the scheduler iterates until it can transfer the control to a ready thread.

The scheduler first checks whether there is a ready thread. If that is the case, the scheduler transfers the control to it. For instance, in Fig. 5 (elements that are either added or removed in the scheduling step are italized, e.g., $th2$) the scheduler checks the queue of ready threads (1). As it is not empty (2), it extracts the first thread in the queue, $th2$ (3-4). Then, the scheduler transfers the control to $th2$ (5).

```
if ¬ activeThread.IsBlocked() then
   readyThreadQueue.Enqueue(activeThread)
else
   it is blocked on a service, a replica, or a lock
endIf
activeThread ← null
while activeThread = null Do
   if ¬ readyThreadQueue.IsEmpty() then
      there are ready threads, choose the first one
      activeThread ← readyThreadQueue.Dequeue()
   else
      get a message from the external queue, the operation blocks if the queue is empty
      msg ← messageQueue.Dequeue()
      if ¬ threadTable.IsRegistered(msg.Tid()) then
         it is the first request from this transaction, a new server thread is created for it
         activeThread ← system.NewServerThread()
         threadTable.Register(msg.Tid(), activeThread)
         the message is added to the corresponding service queue
         activeThread.EnqueueMsg(msg)
      else it is a message for a registered transaction
         if (msg.Kind() = request) or (msg.Kind() = reply) then
            it is a request from a registered client or a reply, get the associated thread
            serverThread ← threadTable.GetThread(msg.Tid())
            serverThread.EnqueueMsg(msg)
            if (msg.Kind() = reply) or serverThread.IsBlockedOn(msg.Service()) then
               the current message unblocks the thread and it becomes the active thread
               activeThread ← serverThread
            elsif serverThread.IsTerminated() then
               the protocol has been violated, the client has issued
               a request after the protocol has finished
               abort current transaction
            else
               the thread is not awaiting this request and hence, it is not unblocked
            endIf
         else it is an abort or commit message
            if (msg.Kind() = abort) or serverThread.IsTerminated() then
               do concurrency control and recovery processing;
               as a result some locks can be released and some threads unblocked
               transManager.ProcessTransTermination(msg.Kind(), msg.Tid(),
                  readyThreadQueue)
               threadTable.Remove(msg.tid)
            elsif serverThread.IsAwaitingRequest() then
               the client has violated the protocol, abort the current transaction
               transManager.ProcessTransTermination(abort, msg.Tid(),
                  readyThreadQueue)
               threadTable.Remove(msg.tid)
            else it is a commit but the thread has not terminated
               annotate in the server thread that the client transaction has terminated,
               if this thread blocks again on a service, the transaction will be aborted
               if the thread ends its execution, the transaction will be committed
               serverThread.ClientHasTerminated()
            endIf
         endIf
      endIf
   endIf
endwhile
system.TransferControl(activeThread)
```

Figure 4. MTRDS **Algorithm**

If the queue of ready threads is empty, the scheduler processes messages from the external queue until there is at least a ready server thread (a new thread is created or a thread is unblocked).

If the extracted message is a request of a registered client, there are several possibilities. The server thread is blocked awaiting (a) this request message, (b) a different message or (c) a lock, or either (d) it has finished its execution. In any of the cases (a-c) the scheduler will enqueue the message in the corresponding service queue of the thread. In case (a) it will transfer the control to it. In case (d) the transaction will be aborted and removed from the thread table. In the cases (b-d) a new message will be processed, as no threads are still ready.

Case (a) is depicted in Fig. 6. In this example, all server threads ($th1$ and $th2$) are blocked, hence, the queue of ready threads is empty (a-b). Then, the scheduler processes the first message ($r1_{T1}$) from the external queue (3-4). As the message comes from a registered transaction (5-6), $T1$, and the corresponding thread ($th1$) is awaiting it (7-8), the scheduler forwards the message (9) and transfers the control to it (10).

	message
Kind	Returns the kind of the message: request, reply, commit, or abort
Service	If the message is a request, this method returns to which service is aimed
Tid	Returns the transaction identifier (tid) of the transaction associated to the message
	readyThreadQueue
IsEmpty	Returns true if the queue is empty
Enqueue	Adds a thread to the queue
Dequeue	Extracts the first thread from the queue
	threadTable
IsRegistered(tid)	Returns true if there is an entry for that transaction
Register(tid, thread)	Registers a client transaction and its associated server thread
GetThread(tid)	Obtains the thread associated to that transaction
Remove(tid)	Remove the entry corresponding to that transaction
	thread
IsBlocked	Returns true if the thread is blocked
IsTerminated	Returns true if the thread has finished its execution
IsAwaitingRequest	Returns true if the thread is blocked waiting for a request
IsBlockedOn(service)	Returns true if the thread is blocked waiting a request for that service
ClientHasTerminated	Annotates that the client transaction has finished
EnqueueMsg(msg)	Enqueues msg in the corresponding internal service queue
	transManager
ProcessTransTermination(abort/commit, tid, readyThreadQueue)	It performs concurrency control and recovery processing relative to an abort/commit operation; it queues unblocked threads in readyThreadQueue
	system
NewServerThread	Starts a new thread
TransferControl(thread)	Transfer the control to that thread
	messageQueue
Dequeue	Extracts the first message from the queue; if the queue is empty the caller is blocked until a message is enqueued by the communication layer

Table 1. Description of object methods

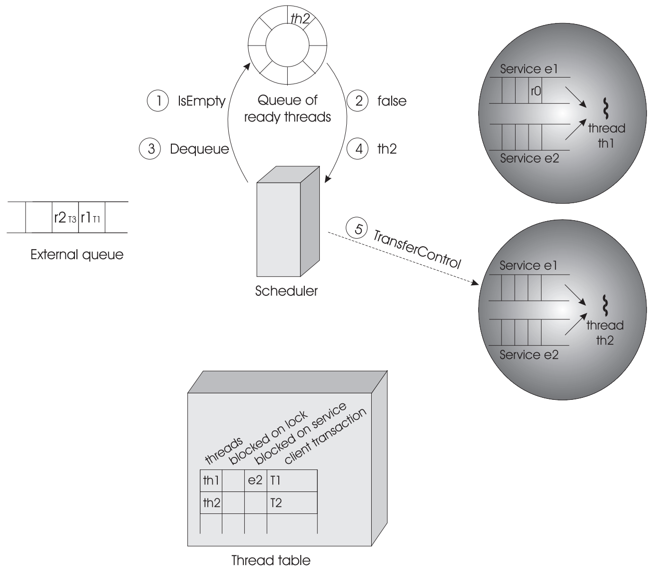

Figure 5. Transferring control to a ready thread

If the message comes from a new transaction, then a new thread is created and an entry with the client transaction and the new thread is added to the thread table. For instance, in Fig. 7, the queue of ready threads is empty (1-2). Hence,

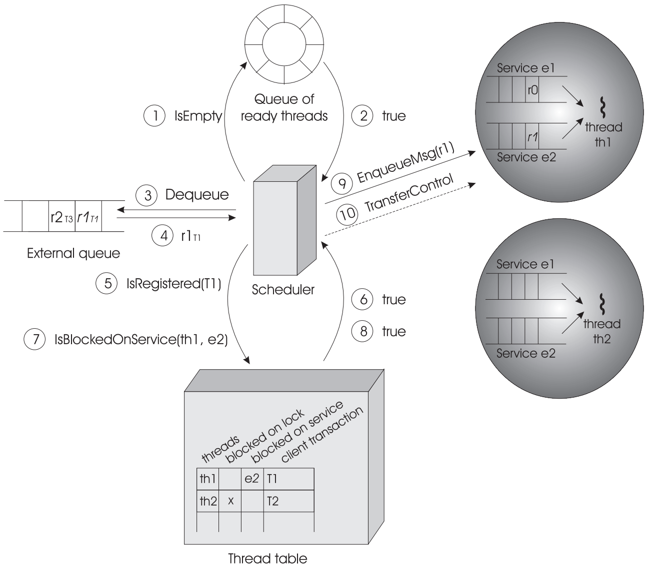

Figure 6. Unblocking a thread

the scheduler takes a message ($r2_{T3}$) from the external queue (3-4). It checks (5) whether the message belongs to a new transaction ($T3$). As this is not the case (6), the scheduler creates a new thread (7), associates the thread with the transaction (8), stores the message in the corresponding queue (9), and transfers the control to it (10).

If the message is a reply to a call to other server, it will unblock the corresponding server thread. The message will be forwarded to the thread and the control transferred to it.

If the message reports a transaction commit, the server thread can be either terminated, ready or blocked. In the first case the transaction is committed. If the server thread is ready or blocked on a lock or awaiting a reply, then it is annotated that the client has finished. If this thread finishes later its execution, the transaction is committed; but if it blocks on a service queue, the transaction is aborted (as the client has not completed the interaction with the server). Otherwise, the thread will be blocked awaiting a request, and therefore, the transaction will be aborted (protocol not completed).

If the message reports a transaction abort, the transaction is aborted.

When the transaction is terminated, either committing or aborting, the thread is removed from the thread table. Both commit and abort of a transaction release the locks held by the transaction. As a result some threads can be unblocked and then they will be queued in the queue of ready threads by the transaction manager.

5. Correctness proof

Definition 1 (Deterministic Scheduler) *Given a sequence of messages, an initial state, and deterministic server code, a deterministic scheduler produces always the same thread interleaving.*

Lemma 5.1 (State Consistency) *Replicas of a multithreaded server implementing the* MTRDS *algorithm have the same state at any scheduling step if the following conditions hold: (1) messages are totally ordered, (2) server thread code is deterministic.*

Proof (lemma 5.1):

We will show that the replicas of a multithreaded server have the same state at a given scheduling point by induction on the number of scheduling steps taken:

1. Induction Basis: At the initial state, when a replica has not received any message yet, all the data structures are empty and the scheduler is blocked waiting for a message in the external message queue. This initial state is the same at all the replicas. When the first message is delivered, the only possible scheduling step is to process this message. As messages are received in total order (condition 1 of the lemma), all replicas will process the same message (the first one in the total order). Additionally, this message can only be the first request of a transaction. Hence, the action taken by all schedulers will be to

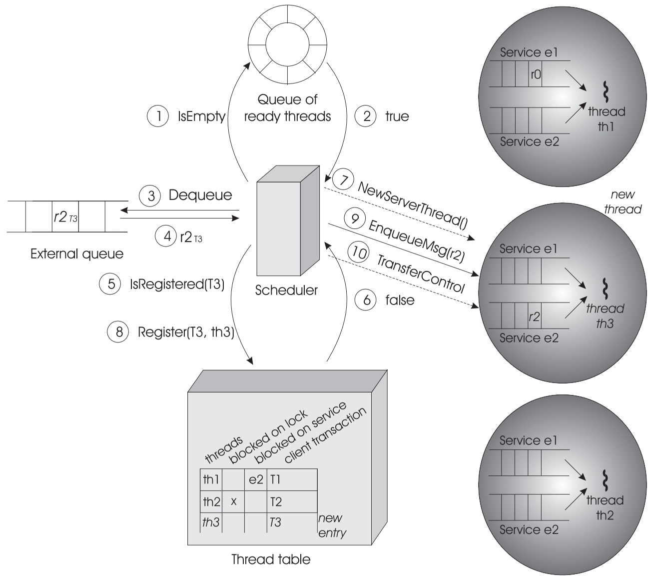

Figure 7. Creating a new server thread

create the associated server thread, to register the new client, to store the message in the corresponding service queue, and to transfer the control to that thread. That is, after the first scheduling step all the replicas have the same state.

2. Induction Hypothesis: Replicas have the same state after the $n-1-th$ scheduling step.

3. Induction Step: Assume that the $n-1$ first scheduling steps have been taken. Now, the active thread will execute until it reaches a scheduling point. As the thread code is deterministic (condition 2 of the lemma) and the state of the replicas is the same (induction hypothesis), all the threads will be in the same state when the control is transferred to the scheduler to perform the $n-th$ scheduling step.

Then, the scheduler performs the $n-th$ scheduling step. Now we will show by means of a case study, once the $n-th$ scheduling step has been made all the replicas have the same state.

(a) There are ready threads. The first ready thread is selected for execution. All the replicas will choose the same thread, as all the replicas have the same state (induction hypothesis).

(b) There are no ready threads, the only possible action is to process a new message from the external queue. This queue might be empty in some replicas and non-empty in others. A replica with an empty queue will block itself until a message is delivered, then it will process the message. A replica with a non-empty queue will directly process the first message in the queue. All the replicas will process the same message, despite differences in the waiting time, thanks to the total ordering (condition 1 of the lemma).

(b.1) If the message is a service request, its treatment will depend on the thread table contents that will be the same at all replicas (induction hypothesis).

(b.1.1) If the client is not registered, all replicas will create a new server thread. A new entry with the transaction and the new thread is stored in the thread table at all replicas. The message is stored in the corresponding internal queue and the control is transferred to that thread at all replicas.

(b.1.2) The client is already registered, and hence, there is a server thread for it. This server thread might be blocked awaiting a request of this kind, awaiting a different kind of request, awaiting a lock release or finished. In the first case, the server thread will become the active thread. In the second and third cases, the message will be queued on the corresponding service queue without unblocking the thread. As the thread will be blocked by the same reason at all replicas (induction hypothesis), all of them will take the same action. In the fourth case, all the replicas will abort the transaction and remove the corresponding entry from the thread table.

(b.2) If the message is a reply (from a call to other server), then it will unblock the calling thread. The message will be forwarded to the thread and the control transferred to it. The same thread will be unblocked at all replicas as they have the same state (induction hypothesis).

(b.3) If the message corresponds to a transaction termination, the associated server thread can be terminated, blocked or ready. The state of the thread will be the same at all replicas (induction hypothesis). The message can be either a commit or an abort.

(b.3.1) If the message is an abort, then the thread is removed from the thread table and the transaction is aborted at all replicas.

(b.3.2) If the message is a commit and the server thread has terminated, the transaction is committed and the thread is removed from the thread table at all replicas.

(b.3.3) If the message is a commit and the server thread is blocked awaiting a request, the client has not completed the protocol. The transaction is aborted at all replicas.

(b.3.4) If the message is a commit and the server thread is not blocked on a service, then all replicas annotate in the thread the fact that the client has terminated.

When a transaction finishes either committing or aborting the associated concurrency control and recovery tasks are performed. The lock information will be updated in the same way at all replicas as all have the same lock information (induction hypothesis) and all have processed the same transaction termination message. As a result, some server threads may be unblocked and moved to the queue of ready threads. As the lock queues are traversed in the same order at all replicas, the same server threads will be unblocked and queued in the queue of ready threads in the same order at all replicas.

All the replicas have the same state after an iteration of the algorithm, and all of them either have chosen an active thread or not. In the latter case the next iterations will start from the same state at all the replicas and the reasoning will be the same as above. The scheduling step will finish by all the replicas choosing the same active thread and all having the same state.

Thus, it has been proven that after the $n-th$ scheduling step all the replicas have the same state. □

Theorem 5.1 (One Copy Serializability) *A replicated server whose replicas run the* MTRDS *algorithm is one-copy serializable.*

Proof (theorem 5.1):

In order to fulfill one copy serializability the replicas must serialize conflicting transactions in the same order. As proven in Lemma 5.1 all replicas have the same state at any given scheduling step, thus they will execute transactions with exactly the same interleaving. What is more, transactions will request locks in the same order at all replicas and hence, they will be serialized in the same order. □

6. Implementation issues

The MTRDS algorithm has been implemented as part of *TransLib* [12], an object-oriented library for distributed transaction processing. *TransLib* is used as run-time support for the programming language *Transactional Drago* [17]. Both *TransLib* and *Transactional Drago* support transactional group servers.

The implementation of *TransLib* has been made in Ada 95 [1] using objects and tasks. Server threads are implemented as Ada tasks and thus, services are task entry points. Client/server communication uses *GroupIO* [8], a group communication library providing reliable total ordered multicast. An interesting feature of this library is that it provides n-to-m communication, that is, when a replicated group performs a call, the library takes care of not delivering duplicate messages.

TransLib is adaptable in the sense that it can be customized to different needs. The concurrency control, recovery and scheduling policies can be defined by the programmer. The scheduling policy decides how and when messages are processed at a server. This policy is encapsulated in a scheduler class. The scheduler is in charge of taking messages from the external queue and forwarding them to the corresponding server threads. In *TransLib* there are predefined schedulers for replicated and cooperative groups. The scheduler for replicated groups implements the MTRDS algorithm. The one for cooperative groups allows more concurrency, as they do not have to behave deterministically. The only requirement of this scheduler is that all server threads of a transaction must choose the same message when they perform a selective reception.

In order to enable reuse of schedulers and programmer-defined scheduling policies it is necessary to uncouple the scheduler from server code and vice versa. To achieve this goal, the scheduler must not know the type of the server nor its interface. But it must still be able to create the right type of server threads, maintain references to them, and forward client requests to them. These requirements have been met by means of two class hierarchies. The *Request* hierarchy (Fig. 8, we use the notation of [6]) encapsulates requests (which kind of service is requested and the associated parameters). The scheduler delegates to instances of this class the server thread creation and the actual interaction with server threads. The *ServerTaskObject* hierarchy (Fig. 9) encapsulates server threads in objects to allow the scheduler to maintain references to them.

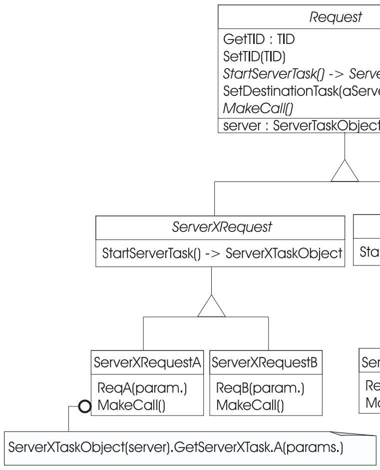

Figure 8. Request hierarchy

The state of a request object contains the service to be called and its parameters. The scheduler can know the identity of the client transaction that has submitted the request by means of the `GetTID` method. Server thread creation is done by means of the `StartServerTask` method.

To be able to delegate to the request object the interaction with the server thread, two methods are needed: `SetDestinationTask` and `MakeCall`. The `SetDestinationTask` method allows the scheduler to inform to the request object about the identity of the server thread to be called. This information is only known by the scheduler. The interaction with the server thread is triggered by the `MakeCall` method.

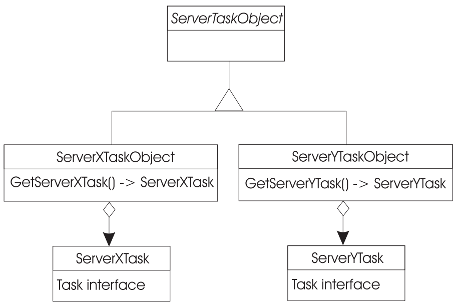

Figure 9. Server task object hierarchy

7. Related work

Some works have focused on fault-tolerance of multithreaded applications. [19] describes a log-based rollback-recovery protocol to resume process execution should a failure occur. They use a combination of checkpointing and logging. Checkpoints contain a snapshot of a process state and additional information to restart its execution from the point at which the state was saved. The log contains information about the occurrence of non-deterministic events before the failure, to reproduce the same conditions during the replay.

The MARS system [11] proposes a solution for determinism in real-time systems with time-triggered event activation and preemptive scheduling. Replica determinism in enforced using timed messages and agreeing on external events. Another approach for real-time systems that solves determinism off-line together with schedulability analysis is [5]. The Delta-4 system proposes a different approach based on semi-active replication following a leader-follower model [2].

There are some works providing CORBA interfaces for object replication like the Eternal system [14] and the object group service (OGS) [9]. OGS guarantees determinism by executing request sequentially in the total order they have been delivered. The Eternal system provides support for replicated multithreaded CORBA servers. In their approach, although replicated servers are multithreaded (using any of the CORBA multithreading models), determinism is enforced by processing RPCs sequentially. During the processing of an RPC, the server can be called from within that RPC (reentrant call), either directly or indirectly. The scheduler detects these situations and allows reentrant calls to be executed, as otherwise, they would produce a deadlock, as there is a single active thread. Indirect calls are tracked down by attaching information to the calls, so the server scheduler can find out that they have been produced by the current RPC under processing, and thus processes them. As pending calls can only be nested reentrant calls a stack is enough to track them down.

There are two important differences between the approach of Eternal and the one of the MTRDS algorithm. First, the Eternal system is based on RPC (the CORBA client/server interaction mechanism) while our algorithm provides a conversational interface. The conversational interface introduces some new difficulties with respect to RPC. With RPC the life of a thread is the life of a single call. However, server threads in MTRDS last for the whole interaction between a client and a server, which usually spans to several calls.

The second main difference is yet more important, and it is that the Eternal system is intended for a non-transactional environment, while the MTRDS algorithm is aimed to a transactional one. In a transactional system it is not possible to process requests one by one, as it is done in the Eternal system. The reason is that during the processing of a request, the active server thread can block itself accessing a conflicting data item, and thus the whole server would be blocked awaiting the release of the lock. This means that more concurrency is needed in a transactional system in the sense that the granularity of the interleavings must be finer. In MTRDS an arbitrary number of threads can be ready at a given time. Additionally, the algorithm needs to take into account the internal messages about transaction management in order to guarantee replica determinism.

Another approach for active replication in database systems is proposed in [15]. This approach is based on an optimistic version of total ordered multicast. It assumes that locks used by a transaction are known in advance, and thus requests can be executed in parallel as far as they do not conflict, thus increasing the concurrency. This proposal provides more concurrency than our approach, as well as a reduced transaction processing latency, taking advantage of the previous assumption. Additionally, each transaction is executed at a single replica, and the rest of the replicas only apply the updates. Thus, this approach provides both availability and performance. On the other hand, the approach is less general than the one presented here, as it requires knowing in advance the locks to be requested.

8. Conclusions

The MTRDS algorithm provides consistent replication of multithreaded transactional servers. Each replica has an attached scheduler that ensures a deterministic behavior of the replica in combination with total order multicast. Replicas are able to process several requests concurrently to prevent the server blocking, as well as to deal deterministically with multiple ready threads resulting from transaction termination. The algorithm has been proposed for servers providing conversational interfaces, but it can be easily adapted for RPC-based interaction (e.g., to use the algorithm in a CORBA environment). Applications that need data consistency and availability can benefit from this approach.

9. Acknowledgments

We would like to thank Rachid Guerraoui for his comments and suggestions. We would also like to thank to the anonymous referees for their helpful remarks.

References

[1] *Ada 95 Reference Manual, ISO/8652-1995*. 1995.

[2] P. Barret, A. Hilborne, P. Bond, D. Seaton, P. Verı́ssimo, L. Rodrigues, and N. Speirs. The Delta-4 Extra Performance Architecture (XPA). In *Proc. of 20th IEEE Symp. on Fault-Tolerant Systems*, pages 481–488, 1990.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, 1987.

[4] K. P. Birman and R. V. Renesse. *Reliable Distributed Computing with Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[5] P. Chevochot and I. Puaut. Scheduling Fault-Tolerant Distributed Hard Real-Time Tasks Independently of the Replication Strategies. In *Proc. of 6th Int. Conf on Real-Time Computing Systems and Applications*, pages 356–363, 1999.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.

[7] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[8] F. Guerra, J. Miranda, Á. Álvarez, and S. Arévalo. An Ada Library to Program Fault-Tolerant Distributed Applications. In K. Hardy and J. Briggs, editors, *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS 1251, pages 230–243, London, United Kingdom, June 1997. Springer.

[9] R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni. System Support for Object Groups. In *Proc. of ACM Conf. Object Oriented Programming Systems, Languages and Applications*, 1998.

[10] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In S. Mullender, editor, *Distributed Systems*, pages 97–145. Addison Wesley, Reading, MA, 1993.

[11] R. D. in Fault-Tolerant Real-Time Systems. *Concurrent Transaction Execution in Multidatbase Systems*. PhD thesis, Technical University of Viena (Austria), 1994.

[12] R. Jiménez Peris, M. Patiño Martínez, S. Arévalo, and F. Ballesteros. TransLib: An Ada 95 Object Oriented Framework for Building Dependable Applications. *Int. Journal of Computer Systems: Science & Engineering*, 15(1):113–125, Jan. 2000.

[13] L. Liang, S. T. Chanson, and G. W. Neufeld. Process Groups and Group Communications. *IEEE Computer*, 23(2):56–66, Feb. 1990.

[14] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications. In *Proc. of the IEEE Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 263–273, Lausanne, Switzerland, 1999.

[15] M. Patiño Martínez, Jiménez Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *In Proc. of Int. Conf. on Distributed Computing, DISC'00. Toledo, Spain*, Oct. 2000.

[16] M. Patiño Martínez, R. Jiménez Peris, and S. Arévalo. Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada. In L. Asplund, editor, *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS 1411, pages 78–89, Uppsala, Sweden, June 1998. Springer.

[17] M. Patiño Martínez, R. Jiménez Peris, and S. Arévalo. Synchronizing Group Transactions with Rendezvous in a Distributed Ada Environment. In *Proc. of ACM Symp. on Applied Computing*, pages 2–9, Atlanta, Georgia, Feb. 1998.

[18] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[19] J. Slye and E. Elnozahy. Supporting Nondeterministic Executions in Fault-Tolerant Systems. In *Proc. of IEEE FTCS-26*, pages 250–259, 1996.