An Algorithm for Deterministic Scheduling of Transactional Multithreaded Replicas and its Correctness *

Ricardo Jiménez-Peris‡, Marta Patiño-Martínez‡, Sergio Arévalo⋆

‡ Technical University of Madrid, E-28660 Boadilla del Monte (Madrid), Spain
{rjimenez, mpatino}@fi.upm.es
⋆ Universidad Rey Juan Carlos, E-28933 Móstoles (Madrid), Spain
sarevalo@escet.urjc.es

Abstract

Middleware platforms are becoming very popular among system developers. Due to its popularity, there is an increasing demand for dependable middleware support. In the past few years several research efforts have concentrated in augmenting the dependability of middleware infrastructures which have led to the definition of the FT-Corba specification. Active replication is one of the main techniques that have been used to achieve some of the required dependability attributes such as high-availability. This kind of replication requires deterministic replicas to behave as a state machine what has been traditionally achieved by restricting replicas to be single-threaded. Unfortunately, single-threading results too restrictive for middleware servers, especially transactional ones, where it is not admissible to process requests sequentially. In this paper, we show how it is possible to remove this restriction. We present a deterministic scheduling algorithm for multithreaded replicas in a transactional framework. Determinism of multithreaded replicas is achieved with a combination of reliable total order multicast and a deterministic scheduler. The former guarantees that all the replicas see the external events in the same order. The latter, ensures that all threads are scheduled in the same way at all replicas. One of the novelties of the approach is that determinism is achieved without resorting to inter-replica communication. Additionally, the paper also addresses how to perform online recovery while maintaining replica determinism in order to keep a high level of availability.

*This paper consolidates results from the papers [1] and [2].

1. Introduction

The increasing popularity of middleware technologies is extending their application areas. However, there is still some reluctance to use this technology in areas with strict dependability requirements. In the last years, several research efforts have focused on increasing the dependability of middleware platforms. A proof of the interest of these efforts is the recent definition of the FT-Corba standard [3]. Some approaches have adopted active replication as a means to attain some dependability attributes in middleware platforms, such as high-availability. Under active replication [4], all the replicas perform the requested services, and hence, site failures can be masked to clients as far as there are operational replicas. In order to ensure replica consistency, all replicas must process the same requests in the same order and behave deterministically. That is, replicas must behave as state machines [4].

The first of these requirements can be achieved using group communication primitives [5, 6], in particular reliable total order multicast. This kind of multicast guarantees that all messages are delivered in the same order at all the available replicas. Deterministic behavior has been traditionally achieved by means of single-threaded replicas. However, this approach of single-threaded replicas might be too restrictive in some environments, like application containers [7], and more especially, those with transactional semantics [1].

One of the open challenges with regard to the underlying theory in active replication is precisely how to cope with the non-determinism that appears naturally due to the typical multi-threaded support provided by modern middleware systems. There have been some early steps to support a restricted form of multithreading for replicated object containers in the Eternal system [7]. In Eternal, active replication is supported by processing requests sequentially. Since using this approach reentrant calls (invocations from an object in a server to the same or a different object in the same server) produce deadlocks, a restricted form of multithreading has been devised. When a replica is blocked on an call, if a reentrant call to the replica is detected, then the new call is processed right away (i.e. instead of processing it in the order it has been received) in order to prevent a deadlock, therefore spawning a new thread. This means that a number of threads can be spawned at a given time, but all of them except the last one will be blocked.

Unfortunately, this technique cannot be extended to a transactional context. In a transactional system a request (service or method invocation) that is part of a transaction can lock data. These data will not be freed until the transaction finishes. If the replica is single-threaded and another request from a different transaction tries to access any of the locked data in a conflicting mode, that request will not finish until the first transaction finishes. Therefore, the replica will be blocked, no matter if there are incoming requests that do not access the same data items.

Another essential issue to achieve high availability is recovery. Since sites fail and the goal of replication is to achieve availability, failed or new replicas should be recovered. Recovery is usually performed off-line to ease obtaining a consistent snapshot of the system state. That is, in order to perform recovery, the system activity is stopped and then an image of the data from an up-to-date replica is transferred to the recovering one. However, this traditional approach contradicts the initial high-availability goal of replication. Recovery in a replicated system should be performed so that the system remains available during recovery or at least the period of unavailability is reduced as much as possible [8].

In this paper, we address these two issues: deterministic scheduling of multithreaded replicas and online recovery. We present a scheduling algorithm for multithreaded replicas in a transactional context extended with an online recovery protocol. The algorithm supports the active replication of multithreaded servers whilst ensuring a deterministic scheduling without any communication among them. Recovery is integrated in the algorithm so that it can be performed in parallel with regular processing and still maintaining replica determinism. We believe this is an important contribution towards obtaining highly available transactional object containers, like the ones of the Corba Object Transaction Service [9] and Enterprise JavaBeans [10].

This research is performed in the context of the ADAPT project[1]: the goal of ADAPT is to provide support for modern middleware so it can adapt itself to the changing environment, for instance, by tolerating failures and performing online recovery. This paper address these goals in the context of multithreaded transactional replicas.

The paper is organized as follows. The assumed system model is described in Section . Then, the different sources of non-determinism are identified in Section . Section presents a deterministic scheduling algorithm for multithreaded replicas (MTRDS) that is extended with online recovery in Section . A discussion of possible extensions of the algorithm are presented in Section . Our approach is compared with related work in Section . Finally, we present our conclusions in Section . The correctness of the MTRDS algorithm is proven in Appendix A.

[1]More information about ADAPT can be found in http://adapt.ls.fi.upm.es/adapt.htm

2. System Model

The system consists of a set of sites $S = \{S_1, S_2, ..., S_N\}$ connected by a network. Sites fail by crashing (no Byzantine failures) and share the same environment (disk space available, etc.). Heterogeneous environments are dealt with in Section . We do not consider the non-determinism introduced by software interrupts. A way to deal with software interrupts is to convert these asynchronous signals to synchronous messages as in Targon/32 [11], or alternatively, they may be queued until the server polls for them as in Delta-4 [12]. The kind of asynchronous signals that might be of interest in a transactional system are timeouts. We show in Section how timeouts can be introduced without loosing determinism. An instance of a server is run at each site to provide fault tolerance and availability. Each instance is called a replica.

2.1 Communication Model

We assume an asynchronous system extended with a possibly unrealiable failure detector [13]. Sites communicate by exchanging messages through reliable channels. Sites are provided with a group communication system [5] supporting virtual synchrony [14]. Group communication systems provide reliable multicast and group membership services. Group membership services provide the notion of view (current connected and active sites). Changes in the composition of a view (addition or deletion) are delivered to the application. We assume a primary component membership [14]. In a primary component membership, views are totally ordered (there are no concurrent views), and for every pair of consecutive views there is at least one member that survives from the one view to the next one. Only a view with a majority of members is allowed to progress (i.e., minority views are blocked). Virtual synchrony ensures that two members that belong to two consecutive views have delivered the same set of messages in the former view. Group communication primitives can be classified attending to the order and reliability guarantees provided [6, 14]. We assume that communication is based on reliable and totally ordered multicast. *Reliability* ensures that a message is delivered to all or none of the available group members. *Total order* guarantees that messages are delivered in the same order to all the group members.

2.2 Transaction Model

Transactions are partially ordered sets of read and write operations [15]. 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[2]. 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 per data item). A transaction can finish successfully (*commits*) or not (*aborts*). In the latter case, the effect is as if the transaction had never been executed. The outcome of a transaction is notified to the application by the underlying transaction processing system.

The correctness criterion for data replication is one-copy serializability [15], which is defined as follows:

Definition 1 (Serial History) *A history H of committed transactions is serial if for any pair of transactions, all the operations of one transaction appear in the history before all the operations of the other transaction.*

Definition 2 (Conflict Equivalence) *Two histories H_1 and H_2 are conflict equivalent, if they are over the same set of transactions and order conflicting operations of committed transactions in the same way.*

Definition 3 (Serializable History) *A history H is serializable, if it is conflict equivalent to some serial history.*

For replicated data, the correctness criterion is one-copy-serializability [15]. That is, replicated data should behave as a single logical copy.

Definition 4 (One-copy Serializability) *A history H upon replicated data is one-copy serializable if and only if it is conflict equivalent to some serial history over all the physical copies.*

Transactions can be distributed. They are started at a site (client) and can access (replicated) data at other sites. Concurrency is allowed within transactions, that is, transactions can be multithreaded [17]. This means that a thread executing a transaction can spawn auxiliary threads within a transaction. All the threads will act on behalf of the transaction.

[2] The algorithm is independent of the kind of locking used. For instance, it would be possible to use semantic locking [16] where the programmer defines lock conflicts exploiting the semantics of the data operations.

Upon completion of a distributed transaction, an atomic commit protocol [18] is run to ensure the atomicity of the transaction. We will assume a standard two phase commit protocol (2PC) [18] that is the one used in current component system platforms like Corba Object Transaction Service (OTS) or Enterprise Java Beans (EJBs). In the 2PC protocol, upon successful completion of a transaction, the coordinator (the initiator of the transaction) multicasts a *prepare* message. The remaining sites participating in the transaction (participants), upon reception of this message, log a *prepared* record, and force their log to disk. A participant, once its log is forced, replies with a *yes* vote. Or if a participant is not able to commit, it replies with a *no* vote. If the coordinator receives a *no* vote, then it sends an *abort* message to all participants. Otherwise, if the coordinator has collected all votes and they are *yes* votes, it logs a *commit* record and forces the log. Once the log is forced it sends a *commit* message to all the participants.

2.3 Client/Server Interaction

A server provides a set of services that clients can invoke. The interaction with servers is conversational [18] and synchronous. That is, a client remains blocked after invoking a server until it gets the reply from the server. A conversational interaction is an interaction in which a client can issue service requests that refer to earlier requests. A server models a business process that spans multiple client requests and has in-memory conversational state. Stateful session beans of Enterprise Java Beans (EJBs) are an example of conversational interaction, where requests correspond to method invocations. A conversational server decides what invocations a client is allowed to do at a given point of the interaction, and knows which results have been produced so far.

One of the advantages of this kind of interaction is that the task of programming a server is simplified as its code only deals with a single client. Furthermore, a server can easily impose a protocol of requests 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. Therefore, 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 invocations). It should be noted that this model of interaction is more flexible than the one provided by stateful session beans.

We assume a thread-per-connection multithreading model (following the terminology from [19]) in which a server thread is spawned for each different client transaction. A server thread executes an instance of the server code and only processes requests from that client transaction.

From now on whenever we use the term thread we will refer to server threads. Server threads are created when the first invocation from a client transaction is processed at a server and are devoted to handle the whole interaction with the client transaction. All server threads at a site (replica) share the server data. A server thread accepts requests explicitly, that is, either accepts a request to a particular service (e.g. like Ada rendezvous) or accepts non-deterministically a request among a specified set by means of selective reception (e.g., like the select statement of Ada or the socket API). An example of selective reception is a server willing to process a request to service e1 or e2 (Fig 1). If there is a request only for one of them, then it is processed. If there are no requests for any of the services, the server thread blocks until a request is received for any of them. If there are requests for both services, one of them is selected and then processed. It is possible to add conditions to the different branches of selective reception. Only branches with conditions evaluated to true are considered for a particular reception.

```
select
  when  condition1 =>
    accept  e1(params.) do
        -- process service e1
    end  e1
or
  when  condition2 =>
    accept  e2(params.) do
        -- process service e2
    end  e2
end select
```

Figure 1: An example of `select` statement

Servers can be replicated to provide fault-tolerance and availability. A replicated server consists of a group of identical servers, or replicas (i.e., with the same code and data), running at different sites that fail independently. Figure 2 shows a two-replica server with two active client transactions, `T1` and `T2`. At each site there are two server threads, one per client transaction. Both server threads have the same code and only process requests (`e1`, `e2`) from their respective client transactions. Server threads at each of the sites, first process a request to service e1, and then a request to e2.

A replicated server can issue requests to other servers (that is, it can also act as a client). These requests are filtered to avoid performing a request multiple times (as many times as available replicas has the caller). Therefore, the effect is as if only one request is made. The reply is sent back to all the replicas. This can be automatically done using a communication library that provides *m to n* group communication [20], or implementing this facility on top of a group

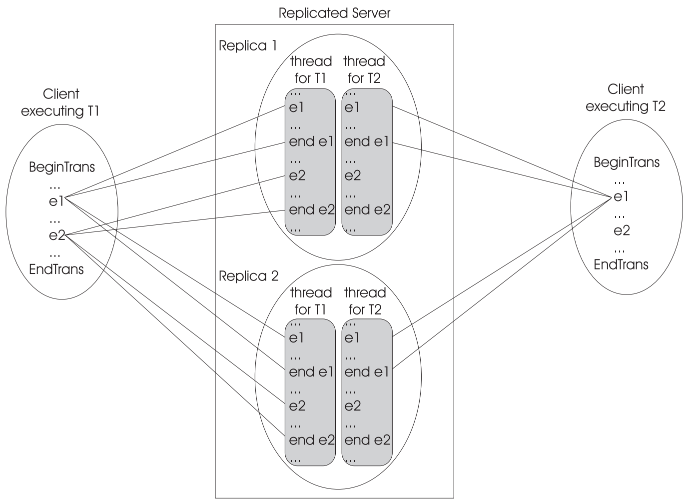

Figure 2: Interaction with a replicated server

communication library [7]. Since request processing is blocking and request acceptance explicit, it is not possible for a server to perform reentrant invocations.

During the first phase of the 2PC protocol, a replicated server votes *yes* when there is a primary component. A minority view does not answer to the prepare message to prevent divergence with the primary component (since it might be running in a different network partition). A replicated server only answers *no* in the advent of a catastrophic failure. This situation is not dealt with in the algorithm and is later discussed in Section .

3 Enforcing Determinism

Active replication requires all replicas to behave deterministically to guarantee their consistency. In this section we study the different sources of non-determinism and how to deal with them.

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 a replica receives. These messages can be either client requests or transaction management messages of the underlying transactional system.

Client requests can reach each site in a different order. Therefore, they can be executed in a different order at each replica loosing replica consistency. The only way to remove this source of non-determinism is to execute conflicting operations in the same order at all replicas. However, there is no general way to know a priori whether two requests will have conflicting operations, thereby all requests (conflicting or not) must be executed in the same order at all replicas. In order to execute operations in the same order, replicas need to receive the same set of requests and process them in the same order. This can be achieved using reliable totally ordered multicast.

Since a server can act as a client of other servers, replies to requests 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. Transaction management messages must also be taken into account to achieve determinism. In particular, transaction termination messages (abort, prepare or commit) can release locks that will unblock transactions (i.e., the threads executing them) blocked on those locks. Therefore, transaction management messages must also be totally ordered to guarantee replica consistency.

However, providing replicas with the same external environment is not enough to guarantee the determinism of multithreaded replicas. Multithreading is itself 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 might produce inconsistencies among the replicas (e.g. serializing two conflicting transactions in different order). For this reason, it is necessary to provide replicas with a deterministic scheduler to ensure that all replicas will perform the same thread scheduling.

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, it can happen that one replica receives 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. 3), while another one has no messages. A deterministic scheduler might decide its next scheduling step as follows: if there are messages to process, take the first one and process it, otherwise transfer the control to a ready thread. Therefore, the scheduler at the replica with messages will decide to process the first pending message, whilst the one with no messages will schedule one of its ready threads. That situation will compromise again the consistency of the replicas despite the fact the scheduler is deterministic.

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 can decide between processing a new message or scheduling a ready thread, it will always do the latter, removing the non-determinism introduced by the non-synchronized reception of messages. There are other deterministic options like alternating the actions of pro-

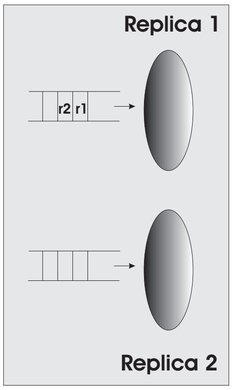

Figure 3: Scenario where total order and deterministic scheduling do not suffice

cessing a new message and scheduling a ready thread. However, this latter option has the drawback that if there are no messages, the replica will be blocked, while it could be processing a ready thread.

The above solution is implicitly using two message queue levels. The *external level* queue corresponds to the communication layer. There is one external queue at each replica. The *internal level* correspond to the services provided by the server. Each server thread has an internal queue per service. Hence, a message in one of these queues has been originated by the transaction associated to that thread and is targeted to the corresponding service.

In Fig. 4 these two levels are shown for a replica providing two services, `e1` and `e2`. The replica is currently running two server threads and each thread has two queues, one for each service. As far as it is possible to progress with the messages in the internal level, no message will be taken from the external queue. Messages taken from the external queue are moved to the corresponding internal level queue. This process is repeated until a thread becomes ready. This two-level queue scheme guarantees that all replicas observe the same set of messages at each scheduling step, since, they only process the minimal prefix of the total ordered sequence of messages needed to progress, which is the same at all replicas.

Selective reception introduces a new source of non-determinism. As clients can be multi-threaded, they can issue concurrent requests to a server. Therefore, each server thread can have an arbitrary number of queued pending requests. When a selective reception is executed, each server thread 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,

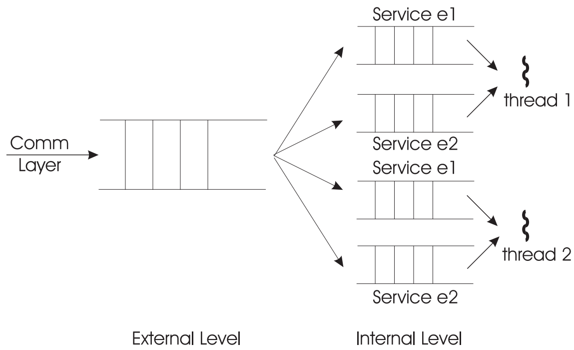

Figure 4: Message queue levels at a replica

and that all 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 replicas will always consider the same set of messages (i.e., the ones in the internal 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.

The support for selective reception enables the application of the proposed approach to an object-oriented setting. For instance, in an object container (or server) with stateless interaction (e.g. stateless session beans), the server creates a server thread upon reception of a request, and then, forwards the message to it. This stateless interaction can be modeled with our approach as follows. Each object class supported by the server will have an associated server thread type. The code of each server thread type will consist of a selective reception statement with as many branches as methods the object class has.

4 The Deterministic Scheduling Algorithm

In this section we describe the *Multithreaded Deterministic Scheduling* algorithm (MTRDS). First, we introduce the data structures used by the algorithm (Section). Section describes how the algorithm works, assuming sequential server threads (i.e., there are multiple server threads running in parallel at a site, but there is only one of them for each transaction). Finally, the algorithm is extended to deal with concurrent server threads working on behalf of the same transaction, that is, server threads can spawn auxiliary threads for additional parallelism (e.g., to perform non-blocking invocations to other servers).

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. We will use an object-oriented notation to describe these data structures. The methods for the different objects are summarized in Table 1.

Messages (msg) are stored temporally in message queues.

As it was discussed in the previous section, these message queues are organized into two levels. The external level (messageQueue) corresponds to the communication layer queue, where messages are totally ordered. This message queue holds client requests, replies to replica requests, and transaction management messages (prepare, commit, and abort). There is a single external message queue per replica. The internal level corresponds to service queues that only hold client requests. There is an internal level per each active transaction (i.e., server thread) at each replica. Extracting messages from any of these queues is a potentially blocking operation. That is, a thread trying to extract a message from an empty internal queue will be blocked until a message is queued. Threads encapsulate their internal message queues.

References to server threads (serverThread) are maintained in thread queues. There are queues for ready threads and for blocked ones. The queue of ready threads (readyThreadQueue), 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.

There is a lock queue associated to each data item. Lock queues contain blocked threads on that data item (threads waiting for a lock to be granted). If the thread under execution, active thread (activeThread), requests a conflicting lock on an item (i.e., an item already locked in a conflicting mode by a different transaction), then 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 (threadTable) is used to keep track of the relation among client transactions and their 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, and the pair of transaction and thread identifiers is stored in the thread table.

Thread status is also kept in the table: finished (i.e. the code of the thread has been completely executed), prepared (i.e., the first phase of 2PC has been completed), waiting for a request (and which request), waiting for a lock (and which lock) or waiting a reply. Termination is annotated to detect the case when a client calls the server once the thread code was completed (a protocol violation).

message	
Kind	Returns the kind of the message: request, reply, prepare, 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(thread)	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
GetTid(thread)	Obtains the tid associated to a thread
Update(thread, blocking_status, resource)	Changes the status of a thread to *ready, blocked on service e, blocked on lock l, blocked on reply, finished, prepared*
Remove(tid)	Remove the entry corresponding to that transaction
AssociateThread(tid, thread)	Associate a thread to a transaction
MarkInTransit(tid)	Mark the transaction as in-transit
IsInTransit(tid)	Returns whether the transaction is in-transit or not
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 (if it has executed a select, it will return true for any service with an open guard in the select)
ClientTerminated	Annotates that the client transaction has finished
HasClientTerminated	Returns whether the client transaction has finished
EnqueueMsg(msg)	Enqueues msg in the corresponding internal service queue
transManager	
ProcessTransTermination(abort/prepare/commit, tid, readyThreadQueue)	It performs concurrency control and recovery processing relative to an abort/commit operation; it queues unblocked threads in readyThreadQueue
system	
NewThread(code)	Starts a new thread that will execute *code* and returns a reference to the created 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

The transaction manager (transManager), for the sake of simplicity, also represents the lock,

```
  -- Check whether it is necessary to perform some processing before changing context
if activeThread ≠ null and activeThread.HasTerminated() then
    -- Has the prepare message arrived for this transaction?
  if activeThread.HasClientTerminated() then
     transManager.ProcessTransTermination(prepare,threadTable.GetTid(activeThread),readyThreadQueue)
    -- else the thread should await 2PC
  end if
elsif ¬ activeThread.IsBlocked() then
    -- the active thread is not blocked, preempt it
  readyThreadQueue.Enqueue(activeThread)
else
  if activeThread.HasClientTerminated() and activeThread.IsAwaitingRequest() then
      -- The client terminated before fulfilling the protocol
    transManager.ProcessTransTermination(abort, threadTable.GetTid(activeThread), readyThreadQueue)
    threadTable.Remove(msg.tid)
      -- else it is blocked on a reply or a lock
  end if
end if
  -- Choose a new active thread
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.NewThread(serverThreadCode)
      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 client has issued a request after the protocol end
          transManager.ProcessTransTermination(abort, msg.Tid(), readyThreadQueue)
          threadTable.Remove(msg.tid)
          -- else the thread is not awaiting this request, so it remains blocked
        end if
      else -- it is a prepare, commit or abort message
        if (msg.Kind() = abort) or serverThread.IsTerminated() then
            -- do concurrency control and recovery processing; as a result some threads might unblock
          transManager.ProcessTransTermination(msg.Kind(), msg.Tid(), readyThreadQueue)
          if (msg.Kind() = abort) or (msg.Kind() = commit) then
            threadTable.Remove(msg.tid)
          end if
        elsif serverThread.IsAwaitingRequest() then
            -- the client has violated the protocol, abort transaction
          transManager.ProcessTransTermination(abort, msg.Tid(), readyThreadQueue)
          threadTable.Remove(msg.tid)
        else -- else it is a prepare but the thread has not terminated
          serverThread.ClientTerminated()
        end if
      end if
    end if
  end if
end while
system.TransferControl(activeThread)
```

Figure 5: MTRDS Algorithm

recovery and transaction managers. The underlying system is represented as an object (system) that provides thread management operations.

4.2 The MTRDS Algorithm

The MTRDS algorithm (Fig. 5) is encapsulated in a scheduler thread at each replica. We follow a multiprogramming approach, that is, at a given time in a site either the scheduler (thread) or a server thread is executing. The scheduler after performing a *scheduling step* transfers the control

($l.$ 39) to a ready thread (that becomes the *active thread*). In this section we assume server threads are sequential.

The active thread returns control to the scheduler (which starts executing from $l.$ 1) when it reaches a *scheduling point*, that is, it accepts a service request, performs a selective reception, requests a lock, invokes a service from other server, ends its execution, enters the prepared state of the 2PC, or finishes the transaction (either committing or aborting).

If the active thread returns control to the scheduler without blocking ($l.$ 4) (e.g., accepting a message from a non-empty service queue or accessing a data item in a non conflicting mode), then the thread is stored in the queue of ready threads ($l.$ 5). Otherwise, the thread will be already blocked (e.g., awaiting a message or a lock). Then, the scheduler iterates until it can transfer the control to a ready thread ($l.$ 11).

The scheduler, first checks whether there is a ready thread ($l.$ 12). If that is the case, the scheduler transfers the control to it ($l.$ 13, 39). Fig. 6 [3] depicts this scenario. The scheduler checks the queue of ready threads (1). Since the queue is not empty (2), the scheduler extracts the first thread in the queue, $th2$ (3-4). Then, the scheduler transfers the control to $th2$ (5).

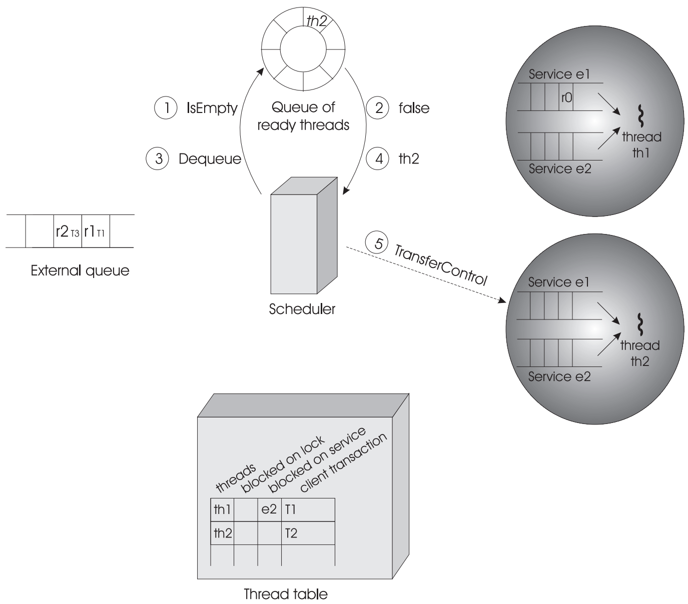

Figure 6: Transferring control to a ready thread

If the queue of ready threads is empty ($l.$ 14), the scheduler extracts messages from the external

[3]In Fig. 6 and subsequent ones, elements that are either added or removed by the scheduler in a scheduling step appear in italics, e.g., $th2$.

queue ($l.$ 15) until there is at least a ready server thread (a new thread is created or one or more threads are unblocked as a consequence of processing a transaction finalization message).

If the message extracted from the external queue is a request ($l.$ 21) from a registered client transaction ($l.$ 20), there are several possibilities. The corresponding server thread is blocked awaiting (a) this message, (b) a different message or (c) a lock, or either (d) the end of the transaction (i.e., it has finished its execution). In any of the cases (a-c) the scheduler will enqueue the new message ($l.$ 23) in the corresponding service queue of the thread. In case (a) it will transfer the control to it ($l.$ 24-25). In case (d) the client has violated the protocol, i.e., it made an invocation after the server thread has already finished its execution. In this case the transaction will be aborted and removed from the thread table ($l.$ 27-28). In the cases (b-d) a new message will be extracted from the external queue ($l.$ 15), since no threads are still ready ($l.$ 11).

Case (a) is depicted in Fig. 7. In this example, all server threads ($th1$ and $th2$) are blocked, hence, the queue of ready threads is empty (1-2). The scheduler extracts the first message ($r1_{T1}$) from the external queue (3-4). Since 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 to it (9), marks the thread as unblocked in the thread table (10), and transfers the control to it (11).

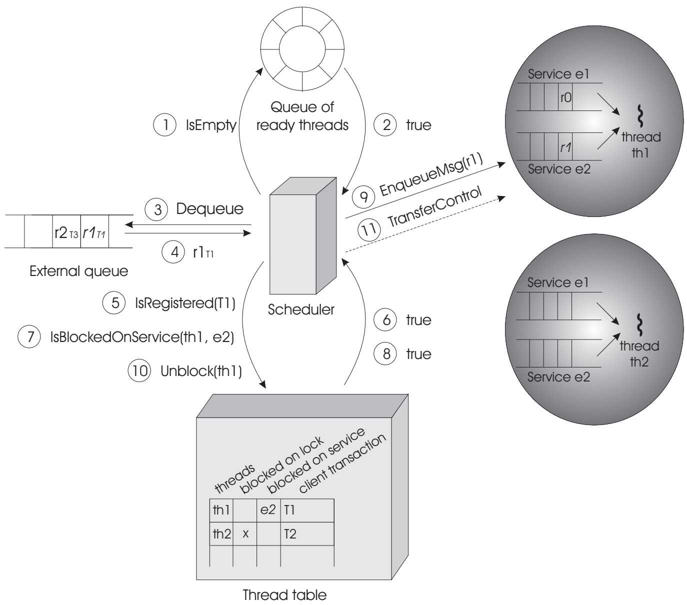

Figure 7: Unblocking a thread

If a message comes from a new transaction ($l.$ 16), then a new thread is created ($l.$ 17) and an entry with the client transaction and the new thread identifier is stored in the thread table ($l.$

18). The creation of a new thread is illustrated in Fig. 8, the queue of ready threads is empty (1-2). Hence, the scheduler takes a message ($r2_{T3}$) from the external queue (3-4). It checks (5) whether the message belongs to a new transaction ($T3$). Since this is 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), marks the thread as unblocked in the thread table (10), and transfers the control to it (11).

If the message is the reply to an invocation to other server (*l.* 24), it will unblock the corresponding server thread (*l.* 25). The message will be forwarded to the thread (*l.* 23) and the control transferred to it (*l.* 39).

If the message extracted from the external queue reports the prepare-to-commit of a transaction (*l.* 29), the server thread can be terminated, ready, or blocked. In the first case (*l.* 30), the thread is ready to commit, and the prepare message is processed consequently (*l.* 31). If the server thread is ready or blocked on a lock or awaiting a reply, then it is annotated in the thread table that the client transaction has started the preparation phase (*l.* 38). If this thread finishes later its execution (*l.* 1-2), the transaction enters the preparation phase (*l.* 3); but if it blocks on a service (*l.* 7, 34), the transaction is aborted (*l.* 8-9, 35-36), since it means that the client has not completed the interaction with the server.

After completing the preparation phase, the server thread is preempted, and eventually a message reporting commit or abort will be delivered. When this message is processed the corresponding action is taken, i.e., commit or abort the transaction (*l.* 29-33).

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. The scheduler will queue them in the queue of ready threads. These activities are performed as part of the `ProcessTransTermination` operation.

4.3 Concurrent Server Threads

The description of the algorithm has been restricted so far to server threads with sequential code. However, the MTRDS algorithm is able to schedule deterministically multiple threads for a given transaction at a replica, and therefore handle concurrent server threads (i.e., server threads that spawn auxiliary threads). Auxiliary threads of a server thread act on behalf of the same transaction. They do not accept service requests, only the server thread is allowed to accept service

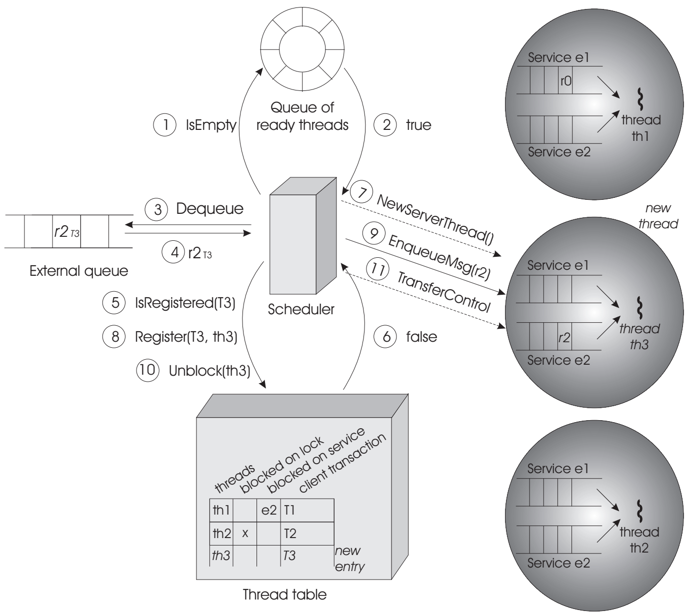

Figure 8: Creating a new server thread

requests. Auxiliary threads are scheduled as any other thread and they can access server data in the same way server threads do (i.e. if an auxiliary thread requests a lock in conflictive mode, only the auxiliary thread is blocked, and it becomes unblocked when the lock is acquired). When an auxiliary thread invokes a service at other server, only it becomes blocked until the reply is received.

Creation of an auxiliary thread just consists in inserting the new auxiliary thread in the queue of ready threads and in the thread table. Each server thread has associated a counter of auxiliary threads. The creation/termination of an auxiliary thread increments/decrements this counter. This counter is useful to determine when a server thread has completely terminated. A server thread terminates its execution only when all its auxiliary threads do so. This means that the `IsTerminated` method for a server thread only returns true when the server thread itself and all its auxiliary threads have terminated.

5 Deterministic Online Recovery

A replicated server is available as far as there is a primary component (i.e., a majority of connected replicas). The primary component can process transactions while some replicas have crashed or are partitioned. Those unavailable replicas should rejoin the primary component in order to maintain the availability of the server when the network connectivity is reestablished or crashed replicas are restarted. Since the primary component might have processed transactions in the meantime, recovering replicas must perform a recovery process before processing new requests. In this process a recovering replica will synchronize its state with the one of the replicas in the primary component (*working replicas*). In what follows, for the sake of simplicity we will assume that there are not overlapping recoveries (i.e. recoveries that start while a recovery process is underway), although we will consider simultaneous recoveries (i.e., recoveries that start in the same view change, for instance, resulting from a partition merge).

One way to implement recovery (state transfer) consists in sending all server data to recovering replicas during the view change [21]. Using this method, it is necessary to either delay the view change processing until all active transactions (*in-transit transactions*) have finished, or include in the state to be transferred information about the exact state of each transaction (program counter, stack, local memory, etc.) to resume the execution of in-transit transactions threads in the recovering replicas. The former solution delays the recovery until all in-transit transactions have finished. The latter solution is technically quite complex, and highly dependant on both hardware and operating system. Additionally, in both solutions processing is suspended during the state transfer, which might take a long time to complete resulting in a considerable loss of availability.

In order to keep a high degree of availability during recovery, it is desirable to perform recovery as less intrusively as possible. That is, transaction processing on working replicas should be disrupted minimally by recovery. Some research has recently been initiated in this direction in the context of replicated databases [8, 22]. However, in that context keeping thread determinism is not an issue. In this section we present how online recovery can be performed deterministically and integrated into the MTRDS algorithm. Section presents how recovery can start without both waiting for in-transit transactions to finish and transferring their whole state. Then, Section shows how to avoid transferring most of the server data during the view change and therefore, reducing the unavailability period. The presented recovery algorithm includes both optimizations.

5.1 Dealing with In-Transit Transactions

State transfer can be performed during the view change for all data that are not write locked by in-transit transactions. After obtaining those data, recovering replicas can start processing new requests that do not conflict with in-transit transactions. However, there is no a priori knowledge (before execution) about which transactions conflict. It could happen that a recovering replica schedules a new transaction that locks some data. Working replicas might schedule an in-transit transaction locking the same data before the new transaction. Therefore, just transferring non write-locked data by in-transit transactions compromises replica determinism and therefore, one-copy serializability. To avoid this scenario a recovering replica also needs to know when in-transit transactions are scheduled, the effects after their scheduling, and the final value (or post-image) of updated data. Based on this information, recovering replicas are not required to run in-transit transactions neither to wait for in-transit transactions to finish to transfer the replica data.

A recovering replica, in order to serialize transactions properly, needs to know which locks were held and requested by each in-transit transaction when the view change took place, and the value of read locked data, since new transactions can read them. That information is transferred during the view change. Since in-transit transactions are not run at recovering replicas, they also need to know the locks that are requested each time an in-transit transaction is scheduled since the view change. In particular, after the preemption of a server thread of an in-transit transaction at a working replica, recovering replicas need to know about the locks acquired by the in-transit transaction server thread, and the status of this thread (ready, blocked on a lock, blocked on a service, blocked on a reply, finished, prepared, aborted). This means that during recovery, in-transit transactions will follow a leader-follower approach [23, 24, 12, 25].

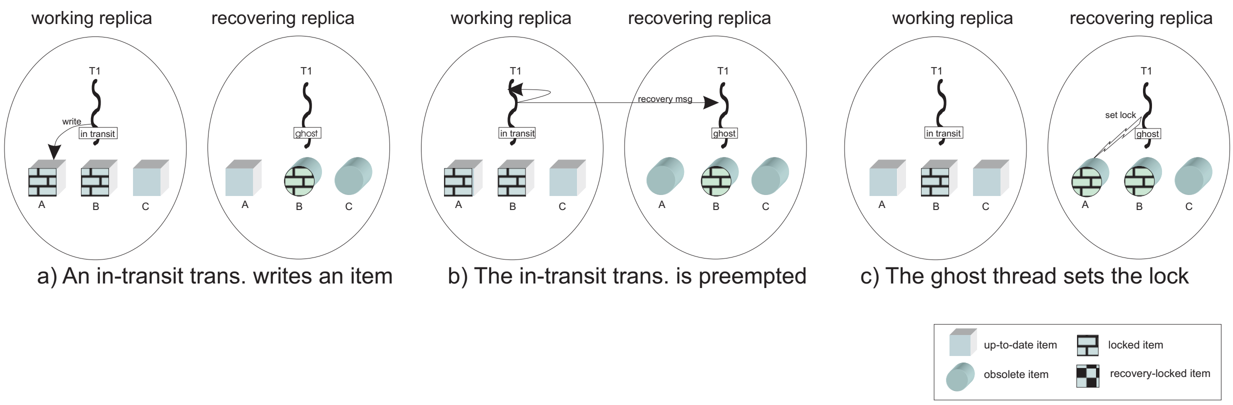

Figure 9: Scheduling of an in-transit transaction

View change messages (with more replicas) are treated as other messages. That is, view changes are queued in the external message queue and they are not processed until they are extracted from that queue. At a working replica, when the scheduler extracts a view change

```
Processing of a view change adding new replicas:
v1:   for each tid in threadTable do
v2:      MarkInTransit(tid)
      end for
v3:   Multicast(recovering_replicas, (scheduler_state,read_locked_data,locks))
v4:   -- Set recovery locks on unlocked data
v5:   activeThread ← system.NewThread(recoveryThreadCode)
v6:   readyThreadQueue.Enqueue(activeThread)

Processing just after preempting an in-transit transaction:
i1:   Multicast(recovering_replicas, (acquired locks, thread state, if blocked on which event))

Prepare message processing:
p1:   serverThread ← threadTable.GetThread(msg.Tid())
p2:   if serverThread.IsAwaitingRequest() then
p3:     -- The client has violated the protocol
p4:     Multicast(recovering_replicas, transaction_data_preimage)
p5:     transManager.ProcessTransTermination(abort, msg.Tid(), readyThreadQueue)
p6:     threadTable.Remove(msg.tid)
p7:   elsif serverThread.IsTerminated() then
p8:     Multicast(recovering_replicas, transaction_data_postimage)
p9:     transManager.ProcessTransTermination(prepare, msg.Tid(),readyThreadQueue)
p10:  else -- The thread has not finished. Commit later
p11:    serverThread.ClientTerminated()
      end if

Abort processing of an in-transit transaction:
a1:   Multicast(recovering_replicas, transaction_data_preimage)
a2:   transManager.ProcessTransTermination(abort, msg.Tid(),readyThreadQueue)
a3:   threadTable.Remove(msg.tid)
```

Figure 10: Recovery code for a working replica

message from the external queue incorporating new sites, the recovery processing starts. The processing of the view change consists in transferring the replica state to the recovering replicas (Fig. 10, *l.* v3) and marking all running transactions as in-transit (Fig. 10, *l.* v1-2). The state consists of the replica data that are not write locked and the scheduler data structures: the thread table, the sequence of transaction identifiers (tids) of the threads in the ready thread queue, and the lock queues. When this information is delivered at a recovering replica, it is copied into the corresponding data structures (Fig. 11, *l.* v1-3), except the sequence of ready tids that is copied into an auxiliary data structure, seqReadyTids. For each in-transit transaction, a thread (*ghost thread*) is created to represent it at the recovering replica. With this purpose, the recovering replica traverses the thread table it has received (Fig. 11, *l.* v4). For each thread in this table, it creates a ghost thread and registers it in the thread table (Fig. 11, *l.* v5-7). Then, it traverses the sequence of ready tids and inserts the corresponding threads in the queue of ready threads (Fig. 11, *l.* v8-10) to recreate the queue of ready threads. Therefore, recovering and working replicas will schedule transactions in the same order.

Ghost threads are used to enforce the same scheduling of in-transit transactions at both recovering and working replicas. That is, when an in-transit transaction is scheduled at a working replica, the corresponding ghost thread will be scheduled at the same scheduling point at the recovering replica (Fig. 9.a). In order to achieve the same effects in all replicas, working replicas send the effects of in-transit transactions to recovering ones every time they are scheduled (Fig. 9.b). When this message is delivered at a recovering replica, the in-transit transaction effects are installed at that replica (Fig. 9.c).

That is, whenever a ghost thread is scheduled, the recovering replica will wait for the effects of the activity of the corresponding server thread from a working replica (Fig. 11, *l.* g1-2). That is, locks acquired by the in-transit transaction, and the current state of the server thread (ready, blocked waiting a request, a reply or a lock, aborted). Schedulers of working replicas will send this information to recovering ones just after the in-transit transaction server thread is preempted (Fig. 10, *l.* i1). Those recovery messages multicast by working replicas are filtered by the underlying communication system guaranteeing that only one is delivered as it happens with regular messages. For the sake of simplicity, all working replicas perform the same recovery actions (it is possible to remove some redundancy, but it makes the protocol more complex). Messages notifying actions performed by in-transit transactions are considered *urgent* and are not queued in the external queue. Instead, they are delivered to the ghost thread immediately (i.e., the recovering replica waits until this message is delivered and does not schedule any other thread in the mean time). Other messages forwarded to ghost threads are just discarded, since ghost threads do not process them. In this way, it is guaranteed that in-transit transactions are scheduled at a recovering replica in the same order as in working replicas. Since locks are requested and released in the same order, the serialization order will be the same at all the replicas.

When a working replica receives a *prepare* message for an in-transit transaction, it sends the updates of the transaction to the recovering replicas (Fig. 10, *l.* p8). Recovering replicas will also receive the *prepare*, and *commit* (*abort*) messages for in-transit transactions. When a recovering replica receives a *prepare* message for an in-transit transaction, it checks if it can commit the transaction (Fig. 11, *l.* p2, p7). In that case, it waits for the transaction updates (post-images) from working replicas (Fig. 10, *l.* p8). The reception of the updates is not a scheduling point and the message is delivered immediately, therefore all replicas will prepare (and commit) the in-transit transactions in the same scheduling step. If the in-transit transaction commits, the updates will be installed. Later, another transaction at a recovering replica can read data written by the committed in-transit transaction.

If an in-transit transaction aborts, working replicas will send the transaction data pre-images to the recovering replicas (Fig. 10, *l.* p4). Recovering replicas will install those pre-images to recover the former values of the data (Fig. 11, *l.* p3-4). Again, this message is a recovery message (and therefore, urgent) and its reception is not a scheduling point.

```
View change processing:
v1:   Receive((scheduler_state,read_locked_data,locks))
v2:   -- Update replica state accordingly including seqReadyThreads
v3:   -- Set recovery locks on unlocked data
v4:   for each tid in threadTable do
v5:     ghostThread ←system.NewThread(ghost_thread_code)
        -- The transaction has already an entry in the table, associate the ghost thread to it
v6:     threadTable.AssociateThread(tid, ghostThread)
v7:     -- Create a transaction with that tid, so it can be committed/aborted
      end for
v8:   for each tid in seqReadyTids do
v9:     readyThreadQueue.Enqueue(threadTable.GetThread(tid))
v10:  end for
v11:  activeThread ← system.NewThread(recoveryThreadCode)
v12:  readyThreadQueue.Enqueue(activeThread)

Ghost server thread code (executed each time the thread is activated):
      -- Wait for the new state of the server thread of an in-transit transaction
g1:   Receive(transaction_state)
g2:   -- Update the ghost thread info in the threadTable and readyThreadQueue accordingly

Prepare message processing (for an in-transit transaction):
      -- Wait for the transaction updates, if transaction commits
      --      or data pre-images, if transaction aborts
p1:   serverThread ← threadTable.GetThread(msg.Tid())
p2:   if serverThread.IsAwaitingRequest() then
        -- The client has violated the protocol
p3:     Receive(transaction_data_preimage)
p4:     -- Update data accordingly
p5:     transManager.ProcessTransTermination(abort, msg.Tid(), readyThreadQueue)
p6:      threadTable.Remove(msg.tid())
p7:   elsif serverThread.IsTerminated() then
p8:     Receive(transaction_data_postimage)
p9:     -- Update data accordingly
p10:    transManager.ProcessTransTermination(prepare, msg.Tid(), readyThreadQueue)
p11:  else -- The thread has not finished. Commit later
p12:    serverThread.ClientTerminated()
   end if

Abort processing (for an in-transit transaction):
a1:   transManager.ProcessTransTermination(abort, msg.Tid(), readyThreadQueue)
a2:   Receive(transaction_data_preimage)
a3:   -- Update data accordingly
```

Figure 11: Recovery code for a recovering replica

It must be noticed that new transactions are scheduled in the same order at both recovering and working replicas. If one of these transactions is granted a lock on a data item, it will also be granted at recovering replicas (the data item is not write locked by the in-transit transaction and therefore, it was transferred during the view change). If a data item was write locked by an in-transit transaction before the view change, the recovering replica does not know its value. If a new transaction access that data item, it will be blocked at both the recovering and working replicas. When the in-transit transaction holding the write lock finishes, the value of the data item will be sent to the recovering replica (the preimage if the transaction aborts, the postimage if the transaction commits), and therefore, the new transaction can access it. With this solution the processing of the view change message is not delayed until all in-transit transactions have finished. The price to be paid is that a message is sent each time an in-transit transaction is scheduled.

```
r1:   for each recovery-locked object, obj, in replica data do
r2:      Send_asynch(obj.GetState())
r3:      Receive(state)
r4:      ReleaseRecoveryLock(obj)
      end for
```

Figure 12: Recovery thread for a working replica

5.2 Increasing Availability during Recovery

Replicas might hold a fairly large amount of data, consequently, transferring all replica data during the view change can be a lengthy operation. In order to perform this data transfer less disruptively, data can be transferred progressively in a sequence of steps. This progressive recovery enables performing transaction execution in parallel with recovery, therefore minimizing the duration of the view change and the associated unavailability period.

In order to obtain a snapshot of the replica data at working replicas and to avoid recovering replicas accessing data that has not been transferred yet, all replicas, working and recovering, set write locks (named *recovery locks*) on all data not locked by in-transit transactions as part of the view change processing (Fig. 10, *l.* v4 and Fig. 11, *l.* v3). Recovery locks prevent access to data that have not yet been transferred to the recovering replicas. This technique has some similarities to the one used in [8], except they wait till all in-transit transactions have finished. Only read-locked data by in-transit transactions are sent to recovering replicas during the view change (besides the scheduler data structures).

The schedulers at both working and recovering replicas create a *recovery thread* (Fig. 10, *l.* v5 and Fig. 11, *l.* v10) to perform recovery in parallel with transaction processing. The code for this thread at the recoverer and recovering replicas is shown in Figs. 12 and 13, respectively. This thread is scheduled as any other thread. At a working replica, the recovery thread is in charge of sending the replica data to recovering replicas (Fig. 12). Each time this thread is activated at a working replica, it sends a data item[4] (Fig. 12, *l.* r2) and waits for its delivery (Fig. 12, *l.* r3). Upon its delivery the working replica releases the recovery lock (Fig. 12, *l.* r4).

At a recovering replica, the recovery thread (Fig. 13) just awaits the reception of this message (Fig. 13, *l.* r2). Upon reception of this message, the recovery thread updates the replica data accordingly (Fig. 13, *l.* r3) and releases the corresponding recovery locks (Fig. 13, *l.* r3).

Only the reception of recovery data is a scheduling point, thereby all the recovery threads go

[4] Data items are sent on a one-by-one basis just for the sake of simplicity. A more efficient approach would be to send them in batches.

```
r1:    for  each recovery-locked object, obj, in replica data do
r2:      Receive(state)
r3:      obj.SetState(state)
r4:      ReleaseRecoveryLock(obj)
       end for
```

Figure 13: Recovery thread for a recovering replica

through the same scheduling points, despite the differences between the code of working and recovering replicas.

Since, the only action that affects determinism is the logical instant at which recovery locks are released, and this is the same at all replicas thanks to the total order, determinism is therefore guaranteed. Furthermore, working and recovering replicas synchronize the logical instant at which transactions are allowed to access data being transferred.

With this recovery protocol the availability of the replica is increased, since the execution of new transactions is not delayed until all the data are transferred. Instead transactions are allowed to progress as far as the data they access have been already transferred to the recovering replicas. A summary of the recovery protocol steps is illustrated in Fig. 14.

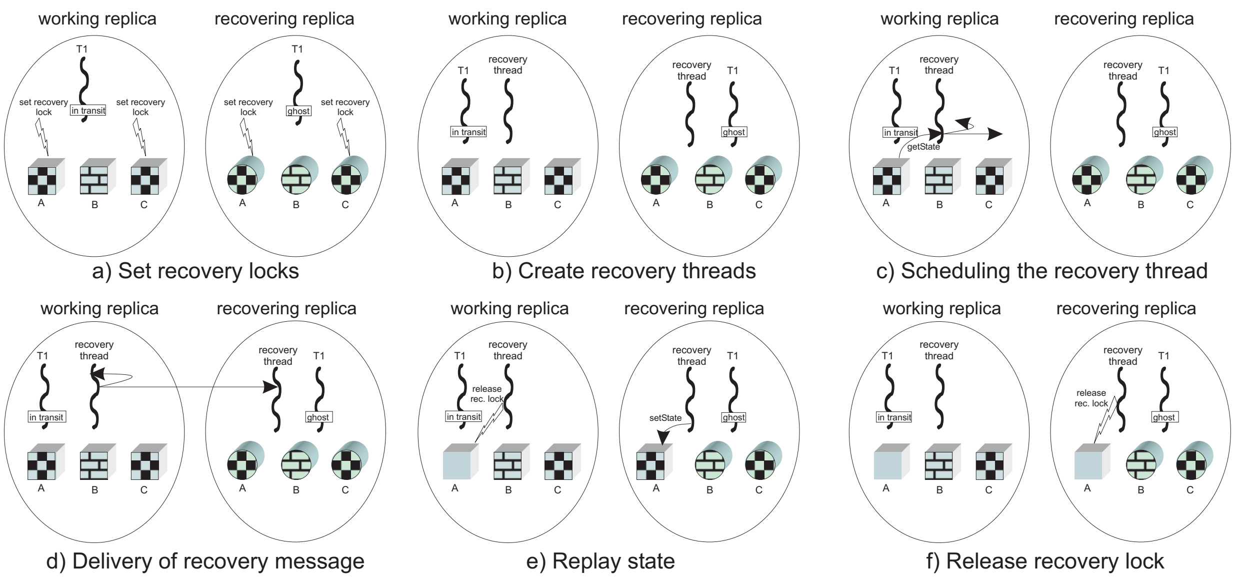

Figure 14: Using a recovery thread

5.3 Recovery from Catastrophic Failures

A catastrophic failure happens when the failure assumptions are violated. In the MTRDS algorithm, this happens when there is not a primary component. Existing minority partitions (if any) become blocked, and the server becomes unavailable.

When a new primary component is available, there is at least one replica that belonged to the last primary component. If that replica has not crashed between the two primary components, the catastrophic failure can be recovered immediately. Any of the non-crashed replicas transiting to the next primary component can transfer its state to the outdated replicas.

However, if all the replicas of the last primary component crashed, it is not enough that any of those replicas belongs to the new primary. Such a replica does not necessarily hold the most up-to-date state. The following scenario might happen in the primary component before the catastrophic failure: a replica crashes, and the other replicas commit a transaction before noticing its failure, and then all of them crash before delivering the new view. In such a situation, it is required to access the state of all the replicas that belonged to the last primary component to obtain the most up-to-date state. Until then, the server will remain unavailable despite there is a primary component. Once this happens, the most up-to-date state is found out by accessing the replica's log and comparing the transaction identifiers among the replicas of the last primary component. One of the replicas holding the highest tid will transfer its state to the rest of the replicas, and then, the server will be available again.

6 Extensions to MTRDS

6.1 Timeouts

Asynchronous signals are not usually needed in transactional software with the possible exception of timeouts. Sometimes it is necessary to limit the duration of transactions to prevent unlimited resource contention by long transactions. Timeouts are an additional source of non-determinism that needs specific mechanisms to be coped with.

The main difficulty to overcome is how all replicas can timeout in a synchronized way. This synchronized timeout mechanism can be achieved as follows. When the first request of a transaction is processed at a replica, a new thread is created to process it. This new thread sets a timeout for that transaction. Since each replica can process the request at a different time, and replica clocks are not perfectly synchronized, they will timeout at different global times. This means that some replicas might decide to abort a transaction due to a timeout and others might be able to commit it before timing out. Determinism can be accomplished if transactions are not aborted right away when the timeout expires. Instead, upon running off a timeout at a replica, the replica will multicast a message to the group notifying the event and resume transaction processing. The delivery of this message can take place before replicas have committed the transaction, or after they have committed it. In the former case, the transaction will be aborted. In the latter case, the timeout message will be dismissed and the transaction will remain committed. Since the taken action, now depends on the ordering of the timeout message with respect to the transaction messages, and this order is the same at all replicas, all replicas will behave in the same way.

One shortcoming of the approach is that if a transaction times out at one replica it will probably timeout at all the other replicas. This means that the timeout of a transaction will surely result in multicasting as many timeout messages as replicas there are. This can be prevented as follows. The role of multicasting the timeout messages can be delegated to a single replica (say the one with smallest view identifier). However, all replicas will annotate the local time at which the first request of a transaction is processed. In case the replica in charge of the timeout fails, a new view change will be delivered, and a new replica will take over the duty of timing out. Additionally, during the view change the new replica in charge of timing out will check which uncommitted transactions have timed out (with respect to its local time) and multicast the corresponding messages notifying it. This technique is similar to the one used to guarantee consistency of timeouts in a replicated commit server [26].

6.2 Dealing with Heterogenous Environments

So far it has been assumed that all replicas share the same environment and that the underlying system (hardware and operating system) behaves in the same way. However, this is difficult, if not impossible to enforce. This means that a mechanism to deal with these situations is needed. The differences in the replica environments can cause that one or more replicas are not able to perform successfully a task while others do.

In general, we can assume that when a piece of work cannot be fulfilled successfully (e.g., write in the disk because the disk is full) an exception is raised notifying it. This means that these environmental differences can result in exceptions raised at some but not all the replicas. This will cause the abortion of the transaction at those replicas where the exception was raised. It can be considered that an exceptional situation that does not happen in a majority is a failure. This failure can be masked by performing some synchronization. The situation can be easily detected and solved at the client API since, it is this API the one in charge of multicasting requests and waiting for replica responses. By extending this API to perform voting on the replica response, replicas that do not provide the same output as the majority will be shutdown and in some cases forced to recover (e.g., in case of exceptions caused by transient failures). This technique has already been taken in previous approaches [27, 25].

6.3 Non-blocking Atomic Commit

The description of the MTRDS algorithm has been made based on the most extended atomic commit protocol, 2PC. However, the MTRDS algorithm can be easily adapted to any other commit protocol. This is important because 2PC is blocking. That is, in some scenarios, when the coordinator fails, some data can remain unavailable until the coordinator recovers. In a highly-available system it would result more convenient a non-blocking atomic commit protocol [28] (e.g. quorum three phase commit, 3PC [29, 30]). The main shortcoming of non-blocking atomic commit is that existing protocols require three rounds of messages in a partially synchronous system in contrast with the two rounds of 2PC. It has been suggested in [31] that lower bounds can be circumvented by using optimistic techniques. Paper [26] shows that the latency of non-blocking atomic commit can be effectively reduced to two rounds by using an optimistic approach (by effectively we mean that aborts are extremely unlikely). This commit protocol can be easily integrated into the MTRDS algorithm in order to provide a higher level of availability without an increased latency.

6.4 RPC-based Client/Server Interaction

The MTRDS algorithm is also applicable to client/server interaction based on RPC. Using RPC as interaction mechanism just simplifies the MTRDS algorithm. The life of a thread under RPC is limited to the duration of the invocation execution. There is no need to track if the request is the one that is expected (since any request is accepted).

7 Related Work

Deterministic execution of multithreaded applications has attracted much attention from different areas, namely, debugging, real-time systems, distributed systems, and more recently middleware platforms. One of the first motivations of deterministic multithreading was debugging [32]. That paper shows how the execution of multithreaded applications can be logged enabling its later replay for the purpose of fault diagnosis.

The field of real-time systems is possibly the one that has shown more interest on deterministic execution [33]. The Delta-4 system [12, 34, 35] was one of the earliest attempts to deal with determinism in the context of real-time systems. The authors propose different replication approaches for real-time applications. Among them, a semi-active replication or leader-follower approach is suggested to deal with non-determinism [23, 24, 12]. In this approach the leader processes all requests, whilst the followers update their state either by processing requests (for deterministic ones) or by notifications from the leader (for non-deterministic ones).

Some more recent approaches deal with multithreading using semi-active (leader-follower) [25] and passive (primary-backup) [36] replication styles. These approaches focus on how to achieve replication in real life applications such as an Apache web server [25] or the Java Virtual Machine (JVM) [36]. Their experimental results show that they can achieve reasonable overheads for those kinds of applications. In these approaches determinism in the presence of multithreading is enforced by recording at a replica (leader or primary) the non-deterministic decisions such as the order in which locks are requested and propagating them to the remaining replicas (followers or backups). Another approach to replicate the JVM under the primary-backup technique has been adopted by [37]. They use the deterministic scheduler approach [1] to reduce the amount of information to be propagated from the primary to the backups. These approaches are attractive for applications with low synchronization requirements. On the other hand, this technique is less attractive when the required synchronization is heavier, that would result in a high overhead and number of messages. For instance, the required synchronization by serializability in a transactional environment is quite high (e.g. a transaction might scan a whole table and all the read locks would be sent to the backups).

The MARS system [38] proposes a solution to enforce determinism in real-time systems based on time-triggered event activation and preemptive scheduling. In this approach, replica determinism in enforced using timed messages and agreeing on external events. [39] is another approach for real-time systems in which determinism is solved off-line in conjunction with the schedulability analysis.

Other research efforts have focused on fault-tolerance of multithreaded applications. [40] addresses the topic of consistency in multithreaded replicas. They use a technique that instead of removing non-determinism is based on logging non-deterministic executions provoked by multithreading and asynchronous events. In the advent of a rollback this log is used to replay a former consistent state of the replica.

There are some systems providing CORBA interfaces for object replication like the Eternal system [7], the object group service (OGS) [41], NewTop OGS [42], AQuA [43], FTS [44], DOORS [45], IRL [46], and reflective Corba [47]. We will focus our comparison on the Eternal system since it is the only one (to the best of our knowledge) that provides support for replicated CORBA containers in the presence of non-determinism such as multithreading. In their approach, they deal with replicated servers that are multithreaded (using any of the CORBA multithreading models), and 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. Otherwise, they would produce a deadlock, since there is only a single active thread. Indirect calls are tracked down by attaching information to the calls, so the scheduler can find out that they have been produced by the RPC under processing, and if this is the case, it is processed. Since 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 (stateful interaction). The conversational interface introduces some new difficulties with respect to RPC. With RPC the life of a thread is the life of a single request. However, threads in MTRDS last for the whole interaction between a client and a server, which usually spans to several requests.

The second main difference, and yet more important, it is that the Eternal system is intended for a non-transactional environment. Eternal processes requests one by one, and only in the case of a reentrant call a new thread is created. This approach although legitimate in a non-transactional environment as the one addressed by Eternal, it is not feasible in a transactional system. The reason of this unfeasibility is that during the processing of a request, the active 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 (e.g., when a transaction holding conflictive locks finishes unblocking several threads that were waiting for the conflictive locks). Additionally, the algorithm needs to take into account the internal messages about transaction management in order to guarantee replica determinism.

In replicated database systems, the problem of non-determinism is overcome by different means. In a replicated database executions do not need to be identical, just equivalent (they deal with data replication instead of process replication). This equivalency is formalized by the concept of one-copy-serializability [15]. In Postgres-R [48] an eager approach for database replication is taken, that is, data modified by a transaction are updated at all replicas within the transaction boundaries. Total order multicast is used to serialize conflicting transactions properly. Papers [49, 50] also deal with eager replication of databases. In those papers replication is performed at the middleware level outside the database.

Online recovery has been studied for replicated databases in the context of the two previous works [8, 22]. [8] discusses online recovery implemented within the database, whilst [22] focuses on how to achieve it at the middleware level. None of these works deal with determinism of multithreaded replicas, since they just pursue data replication.

8 Conclusions

In this article we have shown how to deal with the non-determinism introduced by multithreading in a replicated transactional environment. The MTRDS algorithm provides consistent replication of multithreaded transactional replicas only based on total order multicast. Basing the deterministic scheduling exclusively on total order multicast constitutes an important advantage since replicas do not need any extra messages to preserve the determinism. This is important in application fields where the required synchronization is especially heavy as in transactional environments (where each data access requires synchronization). Replication of multithreaded servers is crucial in modern middleware platforms (e.g., in Corba Object Transaction Service or Enterprise JavaBeans containers). Multithreading enables replicas to process several requests concurrently preventing replica blocking that would result in an unacceptable throughput.

The MTRDS algorithm deals with servers providing conversational interfaces. This allows applying the algorithm to a wider range of applications, for instance, stateful session java beans or entity beans. Additionally, since other interaction models like RPC are particular cases of conversational interfaces, the algorithm can also be applied to them.

Recovery has been integrated into the MTRDS algorithm. The proposed online recovery protocol reduces the period of unavailability of the system because it is run in parallel with transaction processing without loosing determinism. Otherwise, the system would remain unavailable during recovery undermining the initial goal of high-availability pursued by active replication.

In our opinion coping with the non-determinism introduced by multithreading is a necessary step to increase the availability of current middleware technology. We also think that online recovery is crucial to accomplish the high-availability required by modern enterprise applications.

We believe that the algorithms presented in this article can contribute towards achieving high-available middleware platforms.

References

[1] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic Scheduling for Transactional Multithreaded Replicas. In *IEEE SRDS*, 2000.

[2] R. Jiménez-Peris and M. Patiño-Martínez. Online Recovery for Replicated Multithreaded Transactional Servers. In *Workshop on Dependable Middleware-Based Systems*, 2002.

[3] OMG. Fault Tolerant CORBA, OMG TC Document 2000-03-04.

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

[5] K.P. Birman. *Building Secure and Reliable Network Applications*. Prentice Hall, NJ, 1996.

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

[7] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications. In *IEEE SRDS*, 1999.

[8] B. Kemme, A. Bartoli, and O. Babaoglu. Online Reconfiguration in Replicated Databases Based on Group Communication. In *DSN 2001*, 2001.

[9] OMG. *CORBA services: Object Transaction Service (OTS)*. OMG.

[10] Sun. *Enterprise JavaBeans*. http://java.sun.com/products/ejb/index.html.

[11] A. Borg, W. Blau, W. Graetsch, F. Herrman, and W. Oberle. Fault Tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, 1989.

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

[13] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[14] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computer Surveys*, 33(4):143, December 2001.

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

[16] B. R. Badrinath and K. Ramamritham. Semantics-Based Concurrency Control: Beyond Commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992.

[17] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Group Transactions: An Integrated Approach to Transactions and Group Communication. In *Concurrency in Dependable Computing*. Kluwer, 2002.

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

[19] D. C. Schmidt. Evaluating Architectures for Multithreaded Object Request Brokers. *Communications of the ACM*, 41(10):54–60, October 1998.

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

[21] Y. Amir. *Using Group Communication over a Partitioned Network*. PhD thesis, Hebrew University of Jerusalem, 1995.

[22] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-Intrusive, Parallel Recovery of Replicated Data. In *IEEE SRDS*, 2002.

[23] J. S. Banino and J. C. Fabre. Distributed Coupled Actors: A CHORUS Proposal for Reliability. In *ICDCS*, 1982.

[24] P. Barret, A. Hilborne, P. Bond, D. Seaton, P. Veríssimo, L. Rodrigues, and N. Speirs. The Delta-4 Extra Performance Architecture (XPA). In *IEEE FTCS*, 1990.

[25] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. Iyer. Loose Synchronization of Multithreaded Replicas. Technical Report CRHC-02-01, University of Illinois at Urbana-Champaign, 2002.

[26] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and S. Arevalo. A Low-Latency Non-Blocking Atomic Commitment. In *Distributed Computing (DISC'01)*, 2001.

[27] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Exception Handling in Transactional Object Groups. In *Advances in Exception Handling, LNCS-2022*. Springer, 2001.

[28] D. Skeen. Nonblocking Commit Protocols. *ACM SIGMOD*, 1981.

[29] D. Skeen. A Quorum-Based Commit Protocol. In *Proc. of the Works. on Distributed Data Management and Computer Networks*, pages 69–80, 1982.

[30] I. Keidar and D. Dolev. Increasing the Resilience of Atomic Commit at No Additional Cost. In *Proc. of ACM Principles of Database Systems*, 1995.

[31] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults - A tutorial. Technical Report MIT-LCS-TR-821, MIT Computer Science Lab, 2001.

[32] M. Russinovich, Z. Segall, and D. P. Siewiorek. Application transparant fault management in fault-tolerant Mach. In *IEEE FTCS*, 1996.

[33] S. Poledna, A. Burns, A. Wellings, and P. Barret. Replica Determinism and Flexible Scheduling in Hard Real-Time Dependable Systems. *IEEE Trans. on Computer Systems*, 49(2), 2000.

[34] P. Veríssimo, P. Barret, A. Hilborne, L. Rodrigues, and D. Seaton. The Extra Performance Architecture (XPA). In D. Powell, editor, *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer, 1991.

[35] M. Chérèque, D. Powell, P. Reynier, J.L. Richier, and J. Voiron. Active Replication in Delta-4. In *IEEE FTCS*, 1992.

[36] J. Napper, L. Alvisi, and H. Vin. A Fault-Tolerant Java Virtual Machine. Technical Report TR02-56, University of Texas, Dept. of Computer Sciences, 2002.

[37] R. Friedman and R. van Renesse. Transparent Fault Tolerant Java Virtual Machine. Technical report, Computer Science Department, Technion, 2002.

[38] S. Poledna. *Replica Determinism in Fault-Tolerant Real-Time Systems*. PhD thesis, TU Viena, 1994.

[39] P. Chevochot and I. Puaut. Scheduling Fault-Tolerant Distributed Hard Real-Time Tasks Independently of the Replication Strategies. In *RTCSA*, 1999.

[40] J.H. Slye and E. Elnozahy. Supporting Nondeterministic Executions in Fault-Tolerant Systems. In *IEEE FTCS*, 1996.

[41] R. Guerraoui, P. Felber, B. Garbinato, and K. R. Mazouni. System support for object groups. In *OOPSLA*, 1998.

[42] G. Morgan, S.K. Shrivastava, P.D. Ezhilchelvan, and M.C. Little. Design and Implementation of a CORBA Fault-tolerant Object Group Service. In *DAIS*, 1999.

[43] M. Cukier et al. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. In *IEEE SRDS*, 1998.

[44] R. Friedman and E. Hadad. A Group Object Adaptor-Based Approach to CORBA Fault-Tolerance. *IEEE Distributed Systems Online*, 2(7), November 2001.

[45] B. Natarajan, A. Gokhale, S. Yajnik, and D.C. Schmidt. DOORS: Towards High-performance Fault Tolerant CORBA. In *DOA*, 2000.

[46] C.Marchetti, L.Verde, and R.Baldoni. CORBA Request Portable Interceptors: A Performance Analysis. In *DOA*, 2001.

[47] M.O. Killijian and J.C. Fabre. Implementing a Reflective Fault-Tolerant CORBA System. In *Proc. of the IEEE Symposium on Reliable Distributed Systems*, 2000.

[48] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM TODS*, 25(3):333–379, September 2000.

[49] Y. Amir and C. Tutu. From Total Order to Database Replication. In *Proc. of Int. Conf. on Distr. Comp. Systems (ICDCS)*, July 2002.

[50] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Scalable Database Replication Middleware. In *IEEE Int. Conf. on Dist. Comp. Systems (ICDCS)*, 2002.

Appendix A. Correctness

In this section we prove the correctness of the MTRDS algorithm. First, we show that the algorithm guarantees replica consistency without considering recovery. That is, executions in which there are no recoveries. Then, we show that recovery does not introduce inconsistencies and preserves determinism. We assume a homogeneous environment.

Definition 5 (Replica State) *The state of a replica is made out of the replica data (values, and held and requested locks), the scheduler state (i.e., the sequence of server threads in the readyThreadQueue, and the contents of the threadTable), and the state of each thread (current statement, and local state).*

Lemma 1 (State Consistency in the Absence of Recoveries) *In an execution without recoveries, 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 code is deterministic, and (3) replicas start with an empty initial state.*

Proof (lemma 1): We will show that multithreaded replicas have the same state at any given scheduling point by induction on the number of scheduling steps performed:
1. *Induction Basis:* At the initial state, when a replica has not received any message yet, all the scheduler data structures are empty (condition 3) and the scheduler is blocked waiting for a message in the external message queue.

This initial state is the same at all the replicas (condition 3). When the first message is delivered, the only possible scheduling step is to process that message. Since messages are received in total order (condition 1), 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 create the associated server thread, register the new client transaction, store the message in the corresponding service queue, and 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 performed. Now, the active thread will execute until it reaches a scheduling point. Since the thread code is deterministic (condition 2) 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, that once the n-th scheduling step has been performed, all the replicas have the same state.

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

(b) If 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).

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

(b.1.1) If the client transaction 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) There is a server thread for that transaction. This thread might be blocked awaiting a request of this kind, a different kind of request, a lock or it might have finished. In the first case, the 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.

Since the thread will be blocked by the same reason at all replicas (induction hypothesis and condition 1), all of them will take the same action. In the fourth case, all the replicas will abort the transaction because the thread has already finished its execution (the protocol has been violated), and remove the corresponding entry from the thread table.

(b.2) If the message is a reply (from a request 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 due to they have the same state (induction hypothesis).

(b.3) If the message it is related to transaction termination, the associated server thread can be ready, blocked, terminated, or prepared. The state of the thread will be the same at all replicas (induction hypothesis). The message can be: prepare, commit or abort.

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

(b.3.2) It is a prepare message and the server thread has terminated. The preparation phase of the transaction is performed.
(b.3.3) It is a prepare message 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) It is a prepare message and the server thread is not blocked on a service. All replicas annotate in the thread that the client has terminated.
(b.3.5) It is a commit message. The corresponding thread can only be in the prepared state. The server thread is removed from the thread table and the transaction is committed at all replicas.

When a transaction finishes either committing or aborting, the associated concurrency control and recovery tasks (e.g., logging, etc.) are performed. The lock information will be updated in the same way at all replicas, since all of them 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. Since the lock queues are traversed in the same order at all replicas, the same server threads will be unblocked and queued in the ready thread queue in the same order at all replicas. Therefore, all the replicas have the same state after any scheduling step of the algorithm.

All of them have chosen an active thread or none (if there are no ready threads). In the former case the control is transferred to the same active thread. In the latter case, the next iteration of the algorithm will start from the same state at all the replicas and the reasoning will be the same as above. Either the loop will finish choosing the same active thread at all the replicas or the scheduler will be blocked awaiting a new message. In any case the resulting state will be the same at all replicas. Thus, it has been proven that after the n-th scheduling step all the replicas have the same state. □

Lemma 2 (State Consistency from an Arbitrary Common State in the Absence of Recoveries) *Replicas of a multithreaded server implementing the* MTRDS *algorithm keep the same state at any scheduling step of an execution under the following conditions: (1) messages are totally ordered, (2) server code is deterministic, and (3) their state at the start of the execution was identical.*

Proof (lemma 2): The proof is similar to that of Lemma 1 except in that the induction basis assumes an arbitrary state instead of an empty state. □

Lemma 3 (Working Replicas State Consistency during Recovery) *Working replicas of a multithreaded server implementing the* MTRDS *algorithm that start recovery with the same state keep the same state during the recovery process at any scheduling step.*

Proof (lemma 3): Since the recovery thread behaves just as any another thread, with no special treatment, Lemma 2 can be applied here by considering as common state the one they have just after processing the view change message. □

Definition 6 (State Similarity) *A recovering replica is said to have a similar state to the one of a working replica at a given scheduling step, if the following conditions hold:*
(a) Every data item has the same sequence of requested locks at both replicas and fulfills any of the following properties: (a.1) a recovery lock is held on it in both replicas, (a.2) an in-transit transaction holds a write lock on it in both replicas, or (a.3) none of the two previous conditions hold and the data item has the same value in both replicas.
(b) The scheduler state at a recovering replica is similar to the one of a working replica if the two following conditions are fulfilled: (b.1) both readyThreadQueues have the same length and the same sequence of server thread tids; (b.2) both threadTables contain the same set of entries.
(c) The state of the threads at a recovering replica is similar to the one of a working replica if the number of threads at both replicas is the same, both have a recovery thread, and for every server thread in the working replica there is a thread in the recovering replica with the same associated tid that satisfies that, either (c.1) it is a ghost thread, (c.2) it is a server thread with the same state.

Lemma 4 (Recovering Replica State Consistency during Recovery) *A recovering replica of a multithreaded server implementing the* MTRDS *algorithm keeps a similar state to the one of working replicas at any scheduling step during the recovery process.*

Proof (lemma 4): We will prove the lemma by induction on the number of scheduling steps during recovery since the view change:
1. *Induction Basis:* Just after processing the view change message, the state of the recovering replica is similar to the one of the working replicas. This is trivially shown since:
(a) Every data item is either: (a.1) recovery locked, (a.2) write locked by an in-transit transaction, or (a.3) read locked by an in-transit transaction. In case (a.1) and (a.2) data items are similar

by definition. In case (a.3), these data items have been transferred during the view change. Therefore, they have the same value as the ones of working replicas, and are similar.
(b) The scheduler state has been copied into the recovering replica from a working replica during the view change processing.
(c) There is a recovery thread at both the recovering and working replica. The number of ghost threads at the recovering replica is the same as the number of server threads in the working replica, therefore they have a similar thread state.
2. *Induction Hypothesis:* The recovering and working replicas have a similar state after the $(n - 1)$-th scheduling step after the processing of the view change message that included the recovering replica.
3. *Induction Step:* Assume that (n-1) scheduling steps have been performed after the processing of the view change message.

The reasoning for scheduling steps for server threads is the same as in Lemma 1.

Therefore, we will focus on the scheduling of ghost and recovery threads. Any of the following actions will be taken:
(a) Selection of a ready thread. All the replicas will choose the same thread, since all the replicas have a similar state (induction hypothesis).
(a.1) The ready thread is the recovery thread. At working replicas, it will multicast the state of a data item and then it will await the delivery of this message. Since this reception is a scheduling point, a context change will take place. At the recovering replica, the recovery thread will just receive this message and a context change will take place. Then, the recovery thread is annotated as blocked on a reception (waiting for a recovery message) at both recovering and working replicas. The state was similar at the beginning of this step, and since all replicas have changed their state in the same way, the state remains similar.
(a.2) The ready thread corresponds to an in-transit transaction. At working replicas it will perform some activity, possibly setting new locks. The thread will eventually become inactive, either blocking or remaining ready. At this point, the scheduler of every working replica will send any new locks set by the thread as well as the state of the thread (ready, blocked on service s, blocked on a call, blocked on lock l, finished) to the recovering replica. Filtering will guarantee that the message will be delivered only once at the recovering replica. The ghost thread at the recovering replica will await the reception of this message. Once the message is delivered at the recovering replica, this information will be recorded in the corresponding data structures. The scheduling step was started with a similar state. The scheduler data structures of the recovering replica have

a similar state to the one of working replicas at the end of this scheduling step. Data modified by the in-transit transaction at the working replicas are write locked. Since these data are also write-locked at the recovering replica its state is similar to the one of the working replicas.
(b) If there are no ready threads, the only possible action is to process a new message from the external queue.

All the replicas will process the first message from the external queue when it becomes available (the same one thanks to the total ordering).
(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) The client transaction is unregistered. Reasoning like Lemma 1.
(b.1.2) The transaction is registered. Since the recovering replica has a similar state to the one of working replicas, the same actions will be performed as reasoned in Lemma 1. The only difference might be that the transaction is an in-transit one. As shown in (a.2), the recovering replica will yield a similar state to the one of working replicas.
(b.2) If the message is a reply (from a call to other server), then it will unblock the calling thread. At the recovering replica, the thread might be a ghost. In this case it will await the effects from working replicas resulting in a similar state.
(b.3) If the message is related to transaction termination (prepare, commit, abort), the associated server thread can be terminated, blocked or ready. The state of the thread will be the same at all replicas (induction hypothesis).
(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 including the recovering one. If the transaction is an in-transit transaction, the working replica will additionally send to the recovering replica the preimage of the transaction updated data that will be installed by the recovering replica resulting in a similar state.
(b.3.2) If the message is a prepare and the server thread has terminated, the preparation phase of the transaction is performed. If the transaction is an in-transit one, the working replicas after performing the preparation will send the transaction updates to the recovering replica. At the recovering replica the scheduler awaits the reception of the updates from a working replica, then it applies them and performs the preparation. Since the scheduling is the same and the updated data by the transaction too, the resulting state is therefore similar to the one of working replicas.
(b.3.3)-(b.3.4) Similar reasoning as in Lemma 1, taking into account that working replicas will send the state of the thread to the recovering replica.d

(b.3.5) It is a commit message. The thread is removed from the thread table and the transaction committed at all replicas. If the transaction is an in-transit one, then the working replicas will send the postimage of the transaction updated data to the recovering replica, that will install them, resulting in a similar state.
(b.4) If the message carries recovery data, this message will unblock the recovery thread at all replicas, recovering and working. The recovery thread will become active and release the corresponding recovery locks. As a result, some threads might be unblocked. Those threads will be moved to the ready thread queue in the same order at all replicas. Therefore, the state of recovering replicas will remain similar to the one of working replicas.

Therefore, it has been proven that a recovering replica after the n-th step maintains a similar state to the one of working replicas, what proves the lemma. □

Theorem 1 (One Copy Serializability) *A multithreaded server whose replicas run the* MTRDS *algorithm satisties one-copy serializability.*

Proof (theorem 1):

Definition 4 states that in order to fulfill one-copy-serializability all replicas must serialize conflicting transactions in the same order. As proven in Lemma 1, in the absence of recoveries 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 and commit in the same order at all replicas. Hence, transactions will be serialized in the same order. In the advent of a recovery, it has been proven in Lemma 4 that during the recovery the state of working replicas and recovering replicas is similar and therefore, they will serialize transactions in the same order. Upon recovery completion, the state of recovering and working replicas will be the same. Applying Lemma 2 all replicas will maintain the same state at any scheduling step and therefore, transactions will be serialized in the same order. □