Deterministic Scheduling and Online Recovery for Replicated Multithreaded Transactional Servers [*][†]

R. Jiménez-Peris and M. Patiño-Martínez
Computer Science School
Technical University of Madrid
Spain

Abstract

Fault tolerance for middleware platforms has attracted much attention in the last years. A proof of this increasing interest is the recent publication of the FT-Corba standard. Despite the research efforts in this direction, there are still some open questions in this research area. One of them is how to guarantee the determinism of the replicas in the presence of multithreaded executions typical of middleware platforms. Another open problem is to maintain the availability of a replicated server during recovery. In this paper we extend previous work related to the deterministic scheduling of multithreaded replicas with an algorithm for online recovery.

1 Introduction

Availability has become an essential property for today's distributed applications which must continue running despite site failures. Replication is a well-known technique to achieve availability. A proof of the importance of replication is its inclusion in the recent FT-Corba specification [24]. These research efforts have addressed how to implement active replication at the middleware level and more particularly in a Corba environment (Eternal [21], AQuA [8], Electra [19], Arjuna [20], OGS [12], Friends [9], FTS [10]).

In order to provide a certain level of availability, it is necessary to recover failed or partitioned replicas. The recovery of a replica consists in bringing its state up to date, which it is usually accomplished by transferring the state from an active replica. This recovery process implies suspending normal processing until the state transfer is completed. The duration of the recovery process depends on the amount of data to be transferred. Since this data transference can be fairly large, the period of time to perform recovery may be not affordable for applications with high-availability requirements. Given that the system remains unavailable during the duration of recovery, this kind of recovery contradicts the initial high-availability goal of replication. Therefore, a truly high-available system should be able to recover failed replicas whilst the system is online, that is, disrupting regular processing as minimally as possible.

To the best of our knowledge Eternal [22] is the only FT-Corba implementation that deals with how to guarantee replica consistency in the presence of non-determinism such as multithreading. Some recent work has focused on how to attain deterministic recovery in Eternal [23]. In this approach an object is recovered by retrieving an up-to-date state from a running object in quiescent state. This running object waits until is in quiescent state and then transfers its state to the recovering object. During the recovery process the running object involved in recovery does not process requests.

In this paper we propose an online recovery algorithm for replicated transactional servers [15]. In our approach the system does not wait until a replica is quiescent to start recovery what results in a higher availability. The proposed recovery protocol prevents service disruption by performing the recovery of failed replicas in parallel with transaction processing. The paper addresses how to cope with the non-determinism introduced by the multithreaded execution typical of middleware platforms while online recovery is taking place. Some recent papers have studied how to perform online recovery in replicated databases [16, 17] at an internal level [18] and at a middleware level [14]. However, in this

[*]This work has been partially funded by the Spanish Research Council (*CICYT*), contract number *TIC2001-1586-C03-02*.

[†]An extended version of this paper can be found in [13].

context deterministic scheduling of multithreaded computations is not an issue, since the only requirement is one-copy-serializability.

The protocol exhibits two interesting features. First, working replicas do not wait for running transactions to finish before starting the recovery. And second, working replicas do not delay new transactions until the recovery is completed. An important aspect of the recovery protocol lies in that it guarantees that all replicas, working and recovering, behave deterministically despite the multithreaded execution of client requests.

The paper is organized as follows. First, we describe the assumed system model in Section 2. Section 3 introduces the deterministic scheduling algorithm that we will extend with recovery. In Section 4 we present the recovery algorithm. And finally, our conclusions are presented in Section 5.

2 Model

The system consists of a set of sites with persistent and stable storage connected by a network. Sites fail by crashing (no Byzantine failures) and share the same environment (disk space available, etc.). We do not consider the non-determinism introduced by software interrupts. We assume an asynchronous system extended with a possibly unreliable failure detector [6]. Sites communicate by exchanging messages through reliable channels.

Sites are provided with a group communication system supporting virtual synchrony [4, 7] with a primary component membership [7]. We assume that communication is based on a reliable and totally ordered multicast.

A transactional replicated server provides a set of services that clients can invoke. A replicated server consists of a group of identical replicas (i.e., with the same code and data).

The interaction with servers is conversational [11, 26] 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 replicated 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 with this kind of interaction. A server decides what a client is allowed to do at a point of the interaction, and knows which results have been produced so far.

A new server thread is created at each replica for each client transaction. A server thread executes an instance of the server code and only processes requests from that client transaction. Server threads are created when the first call from a transaction is processed at a server. All server threads of a replica share the data of the replica. A server thread accepts requests explicitly, that is, either accepts a request for a particular service (e.g. like Ada rendezvous [1]) or accepts non-deterministically a request among a specified set by means of selective reception (e.g., like the select statement of Ada [1] 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 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.

```
select
    accept  e1(params.) do
        -- process service e1
    end  e1
or
    accept  e2(params.) do
        -- process service e2
    end  e2
end select
```

Figure 1: An example of `select` statement

Figure 2 shows a two-replica server with two transactions, `T1` and `T2`. At each replica 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 replicas, first process a request to service e1, and then a request to service e2. Concurrency control mechanisms (read/write locks) are used to guarantee data consistency.

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 there are at the caller). Therefore, the effect is as if only one request is made. The reply is sent back to all the replicas. Since request submission is blocking and request acceptance explicit, there are not reentrant invocations.

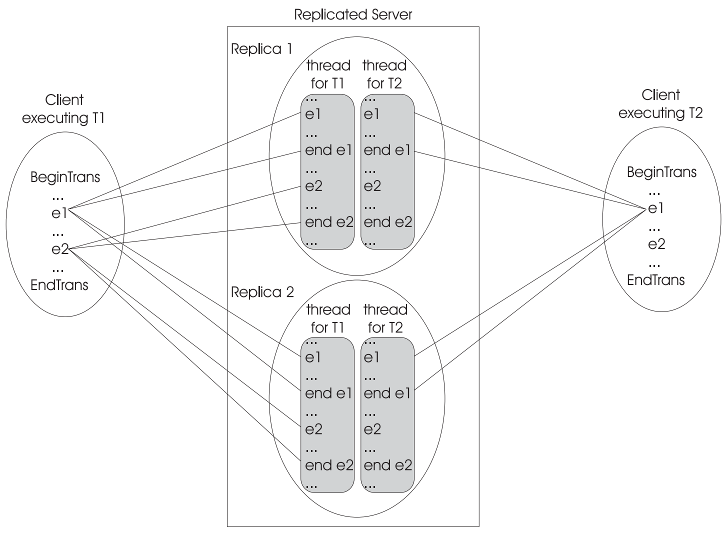


Figure 2: Interaction with a transactional replicated server

3 Deterministic Scheduling

A scheduling is used to guarantee determinism of replicated transactional servers during normal operation. The recovery protocol is integrated with the scheduling protocol. In this section we summarize the assumed deterministic scheduling algorithm. For further details, the reader is referred to [15, 13]

The sources of non-determinism can be classified as external and internal. The 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. This source of non-determinism can be removed by processing requests and replies in the same order at all replicas. In order to execute operations in the same order, replicas need to receive the same set of messages and process them in the same order. This can be achieved using reliable totally ordered multicast.

Replicas also receive transaction management messages. These messages must also be taken into account to achieve determinism. In particular, transaction termination (abort, prepare or commit) messages can release locks that will unblock threads (transactions) blocked on those locks. Therefore, transaction management messages must also be totally ordered to guarantee replica determinism.

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 (let us assume they write the data item x) in the same order can schedule the associated threads in different order. If replica A schedules first request r_1, the associated transaction will obtained the write lock for the item x, whilst in replica B the same will happen for r_2. This will produce a different serialization order of the transactions corresponding to r_1 and r_2 thereby, violating replica determinism. For this reason, it is necessary to provide replicas with a deterministic scheduler which ensures that all replicas will perform the same thread scheduling.

Total ordered requests and deterministic scheduling do not suffice to ensure replica consistency. This is due to total order does not guarantee that all replicas receives messages at the same instant. This means that two replicas could decide to perform different scheduling steps. At a given scheduling point, a replica may have some queued messages, while another one has no messages. The replica with messages could decide to process the first pending message, whilst the one with no messages can only decide to schedule one of its ready threads. That situation compromises the replica consistency. To prevent this situation, a replica will process a new message when it is the only way to progress (i.e., all the threads of the replica are blocked or there are no running threads). Therefore, when a replica has to choose between processing a new message or scheduling a ready thread, it will always do the latter, removing the non-determinism.

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* queues 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 associated transaction and targeted to the corresponding service.

4 Online Recovery

A replicated transactional server is available as far as there is a primary component. 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 [5, 2]. Using this method, it is necessary to either delay the view change until all active transactions (*in-transit transactions*) have finished (i.e. the system is in a quiescent state), 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 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 (if the amount of data is large), what also results 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 as minimally as possible by recovery. In this section, we present how online recovery can be performed in a non-intrusive way, preserving determinism and integrated into a deterministic scheduling algorithm [15]. We first present how recovery can start without both waiting for in-transit transactions to finish nor executing and transferring the whole state of in-transit transactions. Then, we show how to avoid transferring most of the server data during the view change and therefore, reducing the unavailability period.

4.1 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. At the working replicas, an in-transit transaction might be scheduled locking the same data before the new transaction. Therefore, just transferring non-locked data by in-transit transactions compromises replica determinism. A recovering replica needs to know when in-transit transactions are scheduled, the effects after their preemption (i.e., requested locks, ...), and the final value (or post-image) of updated data at commit time. 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 server data.

A recovering replica, in order to serialize transactions properly, also needs to know which locks were held and requested by each in-transit transaction when the view change took place. It will also need the value of read locked data, since new transactions can read them. That information is transferred in the view change.

Since in-transit transactions are not run at recovering replicas. 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 that transaction, 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 an approach similar to the leader-follower approach [3] of Delta-4 [25].

View change messages (with more replicas) are treated as any 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 message from

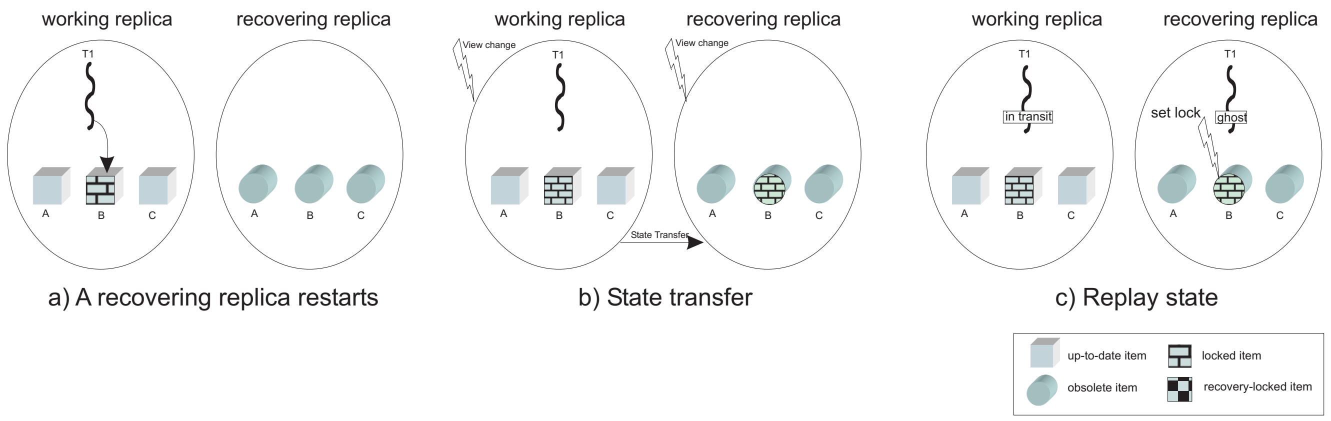


Figure 3: View change processing

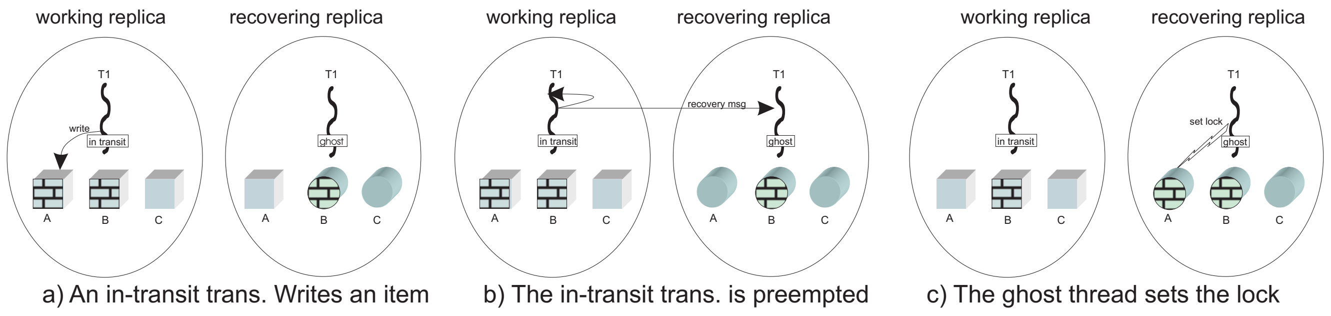
the external queue incorporating new sites, the recovery starts. The processing of the view change consists in transferring the replica state to the recovering replicas and marking all active transactions as in-transit (Fig. 3.b). The state consists of the server data that are not write locked and the scheduler data structures: the external queue, the thread table, the sequence of ready threads in the ready thread queue, and the lock queues. When this information is delivered at a recovering replica, it is copied in the corresponding data structures (Fig. 3.b), except the sequence of ready threads. For each in-transit transaction, a thread (*ghost server thread*) is created to represent it (Fig. 3.c). The recovering replica traverses the sequence of ready threads it has received. For each thread in this sequence, it inserts in its ready thread queue (initially empty) the corresponding ghost thread. Therefore, recovering and working replicas will schedule transactions in the same order.

Whenever a ghost server thread is scheduled, the recovering replica will wait for the effects of the activity of the server thread from a working replica (Fig. 4.a). 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 multicast this information to recovering ones just after the in-transit transaction server thread is preempted (Fig. 4.b). 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. All working replicas perform the same recovery actions to keep determinism and at the same time make the recovery process fault-tolerant. 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 server thread immediately. In this way, it is guaranteed that in-transit transactions are scheduled at a recovering replica in the same way as in working replicas (Fig. 4.c). Since locks are requested and released in the same order, the serialization order will be the same in 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. 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. In that case, it waits for the transaction updates (post-images) from working replicas. 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 at the same logical instant. If the in-transit transaction commits, the updates will be installed. Later, another transaction at a recovering replica can read data written by the in-transit transaction.

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

The advantage of this solution is that recovering replicas can start processing transactions without delaying the view change until in-transit transactions have finished nor executing the whole code of in-transit transactions. The price



working replica
recovering replica
T1
write
in transit
ghost
A
B
C
a) An in-transit trans. Writes an item
recovery msg
b) The in-transit trans. is preempted
set lock
c) The ghost thread sets the lock

Figure 4: Scheduling of an in-transit transaction

to be paid is that a message is sent each time an in-transit transaction is scheduled.

4.2 Increasing Availability during Recovery

Servers might hold a fairly large amount of data, consequently, transferring all sever 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 transaction processing in parallel with recovery, therefore minimizing the duration of the view change and the associated unavailability period, compared to a non-progressive recovery that stops processing until the whole state is transferred.

The only data that are sent to recovering replicas during the view change are in-transit transactions read-locked data. In order to obtain a snapshot of the server data at working replicas and to avoid the access of transactions to data that has not been transferred yet to recovering replicas, all replicas, working and recovering, set write locks (named *recovery locks*) on all data not locked by in-transit transactions (Fig. 5.a) as part of the view change processing.

The scheduler of all the replicas, working and recovering, creates a *recovery thread* to perform recovery in parallel with transaction processing (Fig. 5.b). This thread is scheduled as any other thread. At a working replica, the recovery thread is in charge of sending the server data to recovering replicas. Each time this thread is activated at a working replica, it sends some data and waits for its delivery (Fig. 5.c). Upon its delivery (Fig. 5.d) the working replica releases the recovery lock (Fig. 5.e).

At a recovering replica, the recovery thread just awaits the reception of this message. Upon reception of this message (Fig. 5.d), the recovery thread updates the server data accordingly (Fig. 5.e) and releases the corresponding recovery locks (Fig. 5.f). When a working replica processes that message, it also releases the recovery locks. Only the reception of recovery data is a scheduling point, thereby all the recovery threads go 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 instant 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 transactions are 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.

5 Conclusions

We have presented an online recovery algorithm for replicated transactional servers. This recovery algorithm is used in conjunction with a deterministic scheduling algorithm that ensures replica consistency. The recovery algorithm enables to process client requests in parallel with recovery, what increases server availability during recovery. This contrasts with traditional approaches in which the server should wait until pending transactions have finished to start recovery.

References

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

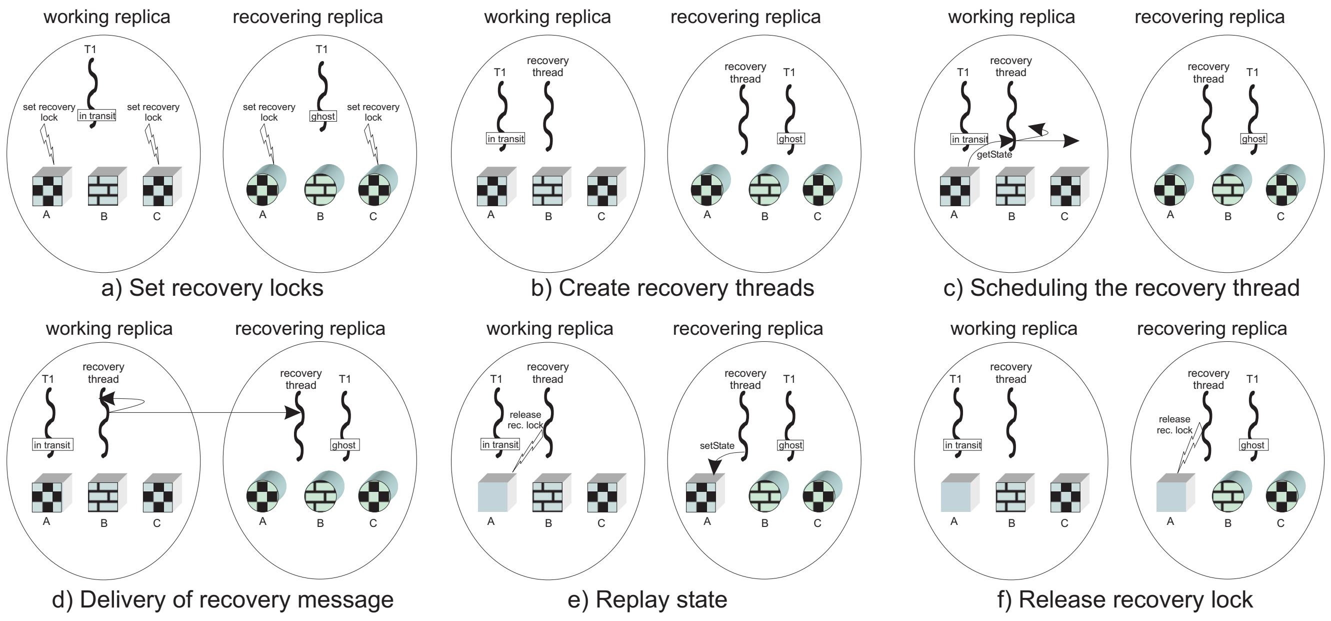


Figure 5: Using a recovery thread

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

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

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

[5] K. P. Birman, T. A. Joseph, T. Raeuchle, and A. E. Abbadi. Implementing Fault-Tolerant Distributed Objects. *IEEE Transactions on Software Engineering*, 11(6):502–508, 1985.

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

[7] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computer Surveys*, 33(4):1–43, Dec. 2001.

[8] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. In *Proc. of IEEE SRDS'98*, pages 245–253, 1998.

[9] J. C. Fabre and T. Perennou. A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS approach. *IEEE Transactions on Computer Systems*, 47(1):78–95, 1998.

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

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

[12] R. Guerraoui, P. Felber, B. Garbinato, and K. R. Mazouni. System support for object groups. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, Oct. 1998.

[13] R. Jiménez-Peris and M. Patiño-Martínez. Deterministic Scheduling for Transactional Multithreaded Replicas. In *Available as technical report at: http://lml.ls.fi.upm.es/ rjimenez/publications/papers/2002/det-sched-tr-2002.pdf*, 2002.

[14] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-Intrusive, Parallel Recovery of Replicated Data. In *Available as technical report at: http://lml.ls.fi.upm.es/ rjimenez/publications/papers/2002/parallel-recovery-tr-2002.pdf*, 2002.

[15] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic Scheduling for Transactional Multithreaded Replicas. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 164–173, Nürnberg, Germany, Oct. 2000. IEEE Computer Society Press.

[16] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, Cairo, Egypt, Sept. 2000.

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

[18] B. Kemme, A. Bartoli, and O. Babaoglu. Online Reconfiguration in Replicated Databases Based on Group Communication. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN 2001)*, Goteborg, Sweden, June 2001.

[19] S. Landis and S. Maffeis. Building Reliable Distributed Systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, Apr. 1997.

[20] G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little. Design and Implementation of a CORBA Fault-tolerant Object Group Service. In *Proc. of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, DAIS'99*, June 1999.

[21] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki. The Eternal System: An Architecture for Enterprise Applications. In *International Enterprise Distributed Object Computing Conference*, pages 214–222, Sept. 1999.

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

[23] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects. In *Proc. of the IEEE Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE Computer Society Press, 2001.

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

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

[26] B. Walter. Nested Transactions with Multiple Commit Points: An Approach to the Structuring of Advanced Database Applications. In *Proc. of 10th Int. Conf. on Very Large Data Bases*, pages 161–171, Singapore, Aug. 1984. Morgan Kaufmann Publishers.