# Preventing Orphan Requests by Integrating Replication and Transactions

Heine Kolltveit and Svein-Olaf Hvasshovd

Norwegian University of Science and Technology

**Abstract.** Replication is crucial to achieve high availability distributed systems. However, non-determinism introduces consistency problems between replicas. Transactions are very well suited to maintain consistency, and by integrating them with replication, support for non-deterministic execution in replicated environments can be achieved. This paper presents an approach where a passively replicated transaction manager is allowed to break replication transparency to abort orphan requests, thus handling non-determinism. A prototype implemented using existing open-source software, Jgroup/ARM and Jini, has been developed, and performance and failover tests have been executed. The results show that while this approach is possible, components specifically tuned for performance must be used to meet real-time requirements.

**Keywords:** Replication, transactions, non-determinism, orphan requests

## 1 Introduction

Fault-tolerance is an important property of real-time and high availability applications. By moving from a centralized to a distributed system, the probability of a total system failure decreases, while the probability of a partial failure increases. A partial failure that is not dealt with correctly could easily jeopardize both consistency and availability of a system.

The *availability* of a system is defined as the fraction of the time that the system performs requests correctly and within specified time constraints [1], while *consistency* is the property that guarantees that the system will behave according to the functional requirements, and applies both to internal (state-changes) and external (output) behavior. Traditionally, transactions are used to ensure consistency [1], while replication provides availability [2]. The two most common types of replication are passive [3] and active [4].

A replicated system must be able to handle the problems occurring due to replicated invocations. A *replicated invocation* is a request from a replicated client to a (possibly replicated) server [5]. The actual problem and solution depends on whether the client is deterministic or not. For deterministic clients, it is only a matter of detecting duplicates, making sure that each request is only executed once and returning the same answer to all duplicate requests.

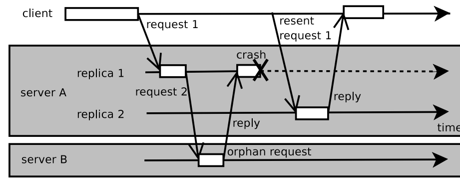


**Fig. 1.** An orphan request caused by non-determinism in server $A$

Many clients, however, are not deterministic. Two common sources for non-determinism are multi-threading and timeouts, but others also exist [6]. These clients are said to be *non-deterministic* and the problem of orphan requests may arise if one of them crashes.

An *orphan request* is a request that is received and processed by a server, but it is no longer valid, normally because of a client failure. Figure 1 illustrates an orphan request. Replica $A_1$ receives a request from the client and invokes service $B$. $A$ is then said to be a *client* of server $B$. $A_1$ then fails before it can reply to the client. Replica $A_2$ is chosen as the new primary and receives the retransmitted request from the client. However, since service $A$ is non-deterministic, the re-sent request 1 might not lead to an identical request 2 to be sent to $B_1$. Consequently, the request from $A_1$ to $B_1$ is an orphan request and its results must be removed. If such a request is not handled, it may cause inconsitencies. Even worse, they might spread to other parts of the system, making the whole system inconsistent.

The main contribution of this paper is an integration of transactions and replication where possible orphan requests caused by non-determinism are aborted. Standard atomic commitment protocols, like 2-Phase Commit (2PC) [7], can not guarantee to remove all orphan requests. The approach suggested by the authors solves this by allowing the transaction manager to break replication transparency and therefore see the individual replicas of the transaction participants instead of the whole replica group. As long as at least one replica of the transaction manager is available it also renders 2PC non-blocking [1, 8]. The problem of orphan requests in replicated systems have been handled before by integrating transactions and replication (e.g. [5, 9, 10]), but these approaches are either ineffective (extra messages in the critical path of the transaction), do not support state in all tiers or make unrealistic assumptions regarding the detection of orphans. The approach adopted in this paper does not have these weaknesses and at the same time it supports checkpointing at any time and restart of crashed replicas.

A prototype based on existing open-source group communication, Jgroup/ARM [11], and transaction implementations, Jini Transaction Service [12], is also presented. The prototype is performance evaluated to see if it can meet the stringent requirements of real-time systems.

The rest of this paper is organized as follows: The system model is presented in Section 2. Section 3 describes other approaches related to the integration of replication and transactions, while supporting non-determinism. A detailed description of how the integration is performed is given in Section 4. The method

developed in Section 4 has been implemented in a test system and the tests performed on this system are presented in Section 5 and discussed in Section 6. Finally, Section 7 concludes the paper.

## 2 System Model

The system consists of a set $S$ of fail-crash processes connected through unreliable channels without network partitions. The processes or *nodes* communicate by sending messages. A group $G$, which is a subset of $S$, implements a service that can be invoked by clients. These may be replicated. A node in $G$ is called a member or *replica* of that service. Group membership is controlled by a *group membership service* that provides an interface for changes in $G$, implements a failure detector, notifies members of changes in $G$ and controls that a request is sent to the correct replica(s) [13]. At any given time members of a group have a *view* of the group which is the set of the agreed upon members. Any replica that crashes is eventually excluded from the view, and any restarted and recovered replica is eventually included in the view.

Passive replication is used, i.e. for each group, there is a primary replica that receives, processes and replies to requests. The state of the backups are updated by periodically performing checkpoints [1], while forced writes of log records are propagated as a part of the atomic commitment protocol. The most common atomic commitment protocol, 2PC, is used.

Replicas are stateful, but have no persistent state. If a replica crashes and is restarted, the state is retrieved from one of the other replicas of the same group. The approach here assumes that there is always at least one replica that has not crashed, therefore the state will never be completely lost. Because there is no persistent state, it is not possible to distinguish a restarted node from a new node. All replicas are assumed to be non-deterministic, thus they can all produce orphan requests.

A request is assumed to be eventually received. They are periodically retransmitted from clients until a reply is received, and duplicates are filtered at each server. Thus, if a primary replica fails and a view with a new primary is installed, the new primary will receive the request when it is re-sent.

## 3 Related Work

Replication and transactions have historically been two separate techniques for achieving fault-tolerance. For instance, CORBA's transaction service (OTS) [14] and replication service (FT-CORBA) [15] are not integrated. A study by Little and Shrivastava [16] looks at two systems, one with transactions and no group communication, and one with group communication and no transactions. Their conclusion is that group communication can be useful for transactions, especially for supporting fast fail-over and active replication.

Many projects deal with replication and transactions, but only a few of these present a proper integration of the two concepts in a non-deterministic environment. Systems that support non-deterministic execution must be able to control its effects. ITRA [17] is an approach that handles the effects by replicating the result of each non-deterministic operation to the backups. ITRA supports replicated transactions by replicating the start, join-operations, prepare (including all operations), commit and abort operations. However, this is not an optimal integration since it incurs an unnecessary high overhead. In our approach only prepare, commit and abort operations are replicated.

Frølund and Guerraoui [18] presents a complete integration of replication and transactions for three-tier applications. However, it supports only stateless middle-tier servers, forcing all state to be stored in the end-tier databases.

Pleisch et al. [5, 19] describes two schemes to handle non-determinism; one optimistic and one pessimistic. The first allows a subtransaction to be committed before its parent, while the latter forces the subtransaction to wait for the commit of the parent. By sending information about how to undo the changes to the backups before invoking a server, orphan subtransactions can be terminated in the pessimistic case, and compensated in the optimistic case. This inserts, however, extra messages in the critical path during failure-free execution.

A CORBA related approach [9] restarts execution of a failed subtransaction on a backup and aborts subtransactions where a parent transaction has failed. This integration, however, assumes that standard distributed commit protocols can be used and does not handle the intricate details of transaction completion in failure scenarios.

## 4 Integration of Transactions and Replication

This section presents an integration of replication and transactions. The goal is to support non-deterministic execution with minimal overhead caused by the integration in a failure-free scenario and minimal change in the application servers (transaction participants).

### 4.1 Replicating the Transaction Manager

To ensure availability, all single points of failure must be avoided. This is especially important for the transaction manager (TM) because it is a central component involved in distributed transactions. If the TM becomes unavailable, the most widely used atomic commitment protocol, 2PC, may block. By using replication 2PC becomes non-blocking [1, 8].

The most important job of the TM is to make the decision to unilaterally abort or commit each transaction. Such a highly critical decision does not favor active replication, since every replica will have to behave deterministically. In practice, TMs are non-deterministic since they rely on timeouts in failure scenarios. This adds non-determinism since it cannot be guaranteed that all replicas timeout at exactly the same time [6]. Also, active replication does not scale well

since executing the same processes on every replica waste resources which could have been used to serve other requests.

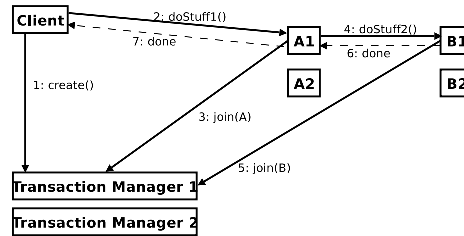


**Fig. 2.** The execution and join phase of a transaction

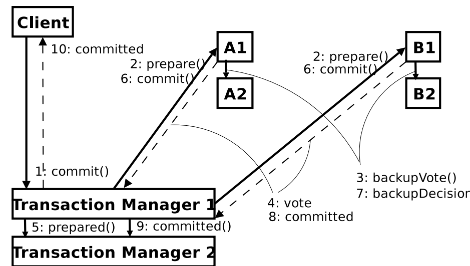


**Fig. 3.** The successful termination of a transaction

As can be seen in Figure 2, the protocol for a passively replicated TM is the same as for the non-replicated before the atomic commitment protocol is initiated by the client in Figure 3. However, a TM that supports 2PC must be able to persistently store the decision to commit or abort the transaction as the final part of the prepare phase [1, 20, 21]. In a non-replicated environment the decision is made persistent by force-writing a record to disk. The round-trip transmission time may be a lot shorter than the time needed to write to the disk. A solution where the prepare decision is persistently saved by sending it to the backups (message 5 in Figure 3) is faster and therefore preferred. In addition, it gives better availability since the prepared transactions can be committed by the backup in case of a primary failure. If a local disk was used, currently prepared transactions may be blocked until the TM has recovered.

A "transaction completed" message is sent to the backups, as indicated by message 9 in Figure 3. This is done instead of the lazy write to the log in the normal non-replicated 2PC [21]. Hence, the replicated nature of the TM is used to provide both availability and persistence of the decision.

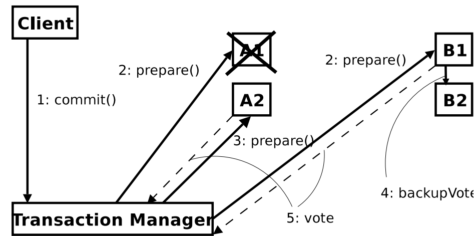


**Fig. 4.** A failure of a primary, and the consequent fail-over

### 4.2 Replicating the Transaction Participants

The transaction participants should be replicated for the same reason as any other component of a distributed system; to avoid single points of failure. To be able to handle non-determinism, passive replication is used.

Passive replication is subject to fail-overs. A *fail-over* happens when a primary replica fails and another replica of the same service is elected as the new primary by the group membership service. Consequently, any re-sent or new requests will be handled by the new primary. If any operation has been executed after the previous checkpoint, the new primary might not be fully updated, and care must be taken to avoid inconsistencies caused by orphan requests. Figure 4 shows a fail-over of a prepare request from $A_1$ to $A_2$.

There are only two ways to cause orphan requests. A failure of the client of the transaction, or a failure of a primary server which acts as a client to another server. The effects of an orphan request can be guaranteed to be removed by aborting all transactions that interact with a replica or client that fails. If the client of the transaction fails, the TM will not receive a commit message from the client. Without the commit message, it can use a timeout to safely abort the transaction. If a primary fails, the TM may still receive a commit message. However, if the TM can determine whether a transaction has been caught in a fail-over or not, it can abort potential orphan requests. The problem is then reduce to the detecting failed primary participants of the transaction.

Normally, replication of a server is hidden from the clients of that service, i.e. *replication transparency*. The unpredictable effects of non-determinism, however, can be controlled at the loss of replication transparency for the TM, by sending the prepare message to the primary only. If a participating primary replica of an active transaction fails, the TM can abort the transaction. Note that it is only the prepare message that does not fail-over. Since the primary persistently stores the vote to the backups during the prepare phase, the backups are then updated and an abort or commit message is allowed to fail-over.

Intuitively, by sending the prepare message only to the primary replicas that joined the transaction, primary failures should be detected: Failed primaries will not be able to reply and the transaction is aborted and possible orphan requests are rolled-back by the transactional abort mechanism. This is true for single fail-overs. Figure 5 illustrates this: When the TM does not get a reply from the

**Fig. 5.** A failure of a primary, without the fail-over

**Fig. 6.** A double fail-over

failed primary replica, $A_1$, it eventually times out and aborts the transaction (as indicated by arrow 5). Contrast this to Figure 4 where the prepare request is allowed to fail-over and the orphan request to service $B$ is not handled.

This protocol has one flaw: If the crashed replica is restarted and a second fail-over back to the original primary occurs, the TM may not be able to notice the failures. This is illustrated in Figure 6. The grey ovals are the current view of the group and the numbers inside are the view identifier. Two fail-overs of group A cause the TM not to notice the fail-over, since the last join message will be identical to the first and looks like a re-send due to a communication error. Therefore, request 2 is an orphan and the transaction should have been aborted. The TM can, however, see the difference of the two join messages if the view identifier was piggybacked on the join message (`join(txn,group,viewID)`, instead of `join(txn,group)`). In the example, the two join messages would have viewID 1 and 4, respectively. Hence, the new protocol will be able to resolve double fail-overs correctly as well.

If a checkpoint was taken after the first request of the transaction, the new primary is aware of the transaction after a fail-over. If the prepare message was sent to the new primary, orphan requests may not be correctly aborted. Figure 7 illustrates this. A checkpoint is take after $A_1$ has joined the transaction. The checkpoint does not include information about request 2, which becomes an orphan request after replica $A_1$ fails. If the TM sends a prepare message to

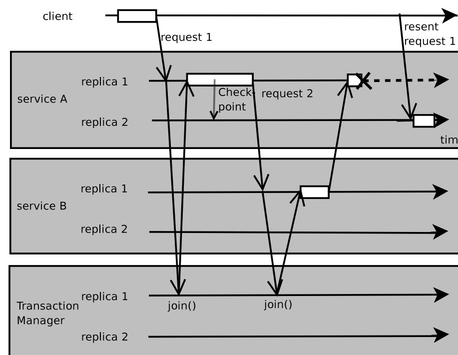


**Fig. 7.** A checkpoint and a possible orphan request in service $B$

replica $A_2$ it would be able to vote yes, and the effects of the orphan request to $B_1$ could cause inconsistencies.

By simply allowing the TM to break replication transparency and therefore be able to avoid the automatic fail-over of the prepare message to a replicated service, it provides a way for orphan requests to be handled easily and correctly. This approach allows checkpointing at any time, thus reducing the time required to bring the state of the new primary up to date and only log records need to be shipped as a part of 2PC. The only requirement for server applications is to implement the 2PC interface. The underlying system adds the view identifier.

### 4.3 Transaction Termination in Failure Scenarios

When the TM is passively replicated as presented in Section 4.1, a transaction may be unable to terminate, therefore blocking other transactions from completing. Consider the following case: A transaction has been created and some or all of the participants have joined it. Then the primary TM fails before the prepare phase has completed. This will leave the new primary with no knowledge of the transaction. When the client asks the TM to commit the transaction, the TM will reply that the transaction is unknown, and the client will assume that it has aborted. However, the transaction participants will still hold their locks on the items accessed by the transaction. Without proper termination of these transactions, the locks could be held forever, blocking other transactions from completing.

The locks held by a failed transaction can be removed by the client if it keeps control of the participants accessed by each transaction. Thus, when the client gets a reply from the TM that the transaction it tried to commit is unknown, the client can abort it. Because of possible nested invocations each participant must be able to tell which other participants it has caused to join the transaction, and so on. However, if one of the participants also fails, the participants invoked by that participant do not get the abort message.

A better way to remove the locks is to use a timeout. Each participant can periodically poll the TM to get the status of each active transaction. If the TM replies that the transaction is unknown, the transaction can be safely aborted. Also, this takes care of the scenario where the client fails.

This approach causes transaction commitment to be non-blocking as long as at least one of the TMs is available. When combined with the avoidance of fail-over for the prepare message and piggybacking the view identifier, all failures are correctly handled to avoid inconsistencies. Potential orphan requests are rolled-back and all types of non-determinism are supported.

## 5 Implementation and Testing

This section gives an overview of the prototype implementation, as well as the environment used for testing and the results of tests executed on the prototype. A presentation of the prototype is given in Section 5.1 and the environment for testing and the tests executed are presented in Section 5.2.

### 5.1 Prototype Implementation

A prototype of the transaction manager that do not allow fail-over of the prepare request was implemented, along with the transaction participants. The servers were implemented as Jini [22] services, and replicated using the Jgroup/ARM system [11].

The system where the tests are executed consists of four conceptual entities: A client, a transaction manager (TM) and two different banks. The banks and the transaction manager are implemented as Jini services that can be discovered and registered by the Jini registry, *Reggie* [12], or the group-enabled registry, *Greg* [23]. The transaction manager is based on the non-replicated *Mahalo* [12] as well as the actively replicated *Gahalo* [24].

### 5.2 The Test Environment

Figure 8 shows the system model. The grey ovals represent entities, while the white boxes are nodes where replicas of servers or the client execute. A single physical node may execute more than one service. The arrows in the figure represent the direction of the invocations.

The life cycle of the transaction used for testing is as follows:

1. A transaction is initiated by the client, and created by the TM.
2. The client invokes the withdraw operation of $\text{Bank}_A$, which joins the transaction.
3. The client invokes the deposit operation of $\text{Bank}_B$, which joins the transaction.
4. The client initiates 2PC, which is controlled by the TM.

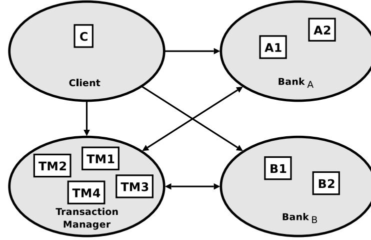


**Fig. 8.** A model of the system used for testing

As modeled in the figure, the TM can have up to four replicas, and the two banks can have up to two replicas each. These limitations are due to the fact that there were only five nodes available for executing the tests.

A Dual AMD MP 1600+ running at 1.4GHz powered each node. A 100Mbit Ethernet connected them and each had 1024 MB of RAM. The tests were executed using Linux kernel 2.6 and Java version 1.5.0.

All tests were carried out by executing 500 transactions and measuring the elapsed time at the client between transaction initiation and transaction completion. This is referred to as the *response time* of a transaction. Similarly, the response time of an invocation is the time passed between calling the remote method of the client and the return of the method call.

Failure-free performance tests were performed using the following configurations:

1. A non-replicated transaction manager and non-replicated banks.
2. Two passively replicated transaction managers and non-replicated banks.
3. Three passively replicated transaction managers and non-replicated banks.
4. Four passively replicated transaction managers and non-replicated banks.
5. A non-replicated transaction manager, with two replicas of each bank.
6. Two passively replicated transaction managers, with two replicas of each bank.
7. Three passively replicated transaction managers, with two replicas of each bank.

In addition, the response times during fail-overs were measured.

## 6 Comparing the Test Results

Table 1 summarizes the results of the test runs made in Section 5.2. The response time averages and standard deviations are presented[1]. It should be noted that these numbers only apply for these test runs and they should not be interpreted as any general response time guarantee, but rather as properties of the specific

[1] The first 50 transactions of each test run are disregarded in this discussion because of extra startup cost.

| Test run | Description | Average (ms) | Standard Deviation (ms) | Delay (%) |
|---|---|---|---|---|
| *Nonreplicated system* | | | | |
| 1 | 1 TM and 2 banks | 47 | 10 | 0 |
| *Passive replication of the TM* | | | | |
| 2 | 2 passive TMs and 2 banks | 77 | 17 | 64 |
| 3 | 3 passive TMs and 2 banks | 92 | 19 | 96 |
| 4 | 4 passive TMs and 2 banks | 106 | 21 | 126 |
| *Fully replicated system* | | | | |
| 5 | 1 TM and 2x2 banks | 75 | 16 | 60 |
| 6 | 2 passive TMs and 2x2 banks | 148 | 25 | 215 |
| 7 | 3 passive TMs and 2x2 banks | 164 | 27 | 249 |

**Table 1.** A summary of the response times for the test runs in Section 5.2

test run. However, they can be used as a reference for comparisons between the individual test runs.

The response time of a transaction was measured at the client and is the time elapsed from transaction creation to transaction termination.

The following sections presents a summary of the results from the test runs and compares passive replication and the non-replicated case. Finally, the fail-over delay is examined.

### 6.1 Cost of Replication

Replication increases the overhead of a service. The results of test runs 1–4, as presented in Table 1, clearly support this assumption. The average response time degrades when adding more replicas of a transaction manager, and the variance of the results increase. The numbers seem to indicate that replicating the TM causes about 50 percent longer response times, while each added replica on top of that increases the response time of about 30 percent of the non-replicated case.

The standard deviation seems to change similarly to the average response time. Table 1 shows a significant leap for the deviation of the response times when the TM is replicated and then scales linearly when adding the third and fourth replica.

Replication of the transaction participants (test run 5) has similar effects as when only replicating the transaction manager (test run 2). There is a 50 percent increase in the response time and about the same for the standard deviation. For test runs 6 and 7 the overhead increases more. Replicating the TM as well (test run 6) doubles the average response time. The cost of executing a fully replicated system with 2 replicas of each server is three times higher than executing a non-replicated one. If 3 replicas of the TM are executed (test run 7), the response time is three and a half times higher than in the non-replicated case.

A closer inspection reveals that for the fully replicated case (test run 6) the group management threads and layers causes an overhead that delays around 60

percent of the invocations, usually for around 10 – 20 ms. The average delay for a transaction just by running the group management threads was found to be 40 – 50 ms. The time to update the backups was found to be around 22 ms for the commit decision at the TM and 12 ms for each of the other updates. When added together these contribute to an average response time of 150 ms which differs with only 1.3% from the measured total time.

To make the non-replicated case fault-tolerant the log could be force written to disk instead of updating the backups. To force write a record to the disk takes approximately 20 ms. A successfully committed transaction requires three log forces and one lazy log write as part of 2PC [21]. Thus, the completion time for a fully fault-tolerant non-replicated system has a response time of around 107 ms. However, this solution does not provide high availability since it has single points of failure.

### 6.2 Fail-over Delay

The observed client-side fail-over delay for the transaction test was found to be as much as 360–490 ms. However, the fail-over delay for a simpler application running on top of the same system was found to be between 200 and 250 ms. These measurements are closer to the real time between a failure and the continuation of the service by a new primary.

Gray and Reuter [1] distinguish five classes of transaction-oriented computing, with various properties and requirements. According to this classification the fail-over delay found here will be sufficient for batch processing, time-sharing (not widely used anymore), client-server and transaction-oriented processing. The last class, real-time processing, however, will probably require client-observed fail-overs of less than 200 ms, depending on the application.

## 7 Conclusion and Further Work

Many applications require high availability and strong consistency. Since system components fail from time to time, a system must be able to tolerate faults. Well known fault-tolerance techniques include transactions and replication. They are widely used and extensively studied as separate concepts and their efficiency has been well proven. However, to support both availability and consistency in a non-deterministic environment, the techniques should be integrated.

This paper addresses the issue of integrating replication and transactions without enforcing replica determinism. This is a highly desirable property since it allows any kind of application to be built on top of the system. The main contribution is an approach where the transaction manager is allowed to break replication transparency to ensure that no orphan requests survive. Thus, full support for non-determinism in general is achieved.

Tests were performed on a prototype built using existing open-source software. The tests show that transactions can be executed in a passively replicated environment with a 200 percent response time increase. Also, a failure of the

primary will cause a fail-over delay of about 400 ms on average for the transaction manager. Measurements on a smaller application, however, indicate that the real fail-over time is probably closer to 200 ms. While these results show that the approach is possible, real-time systems have more stringent performance demands. The response time for a transaction in this prototype is too large for most real-time systems, e.g. telecommunications [25]. However, this is a property of this specific implementaion and not the approach, since both Jini and Jgroup are not tuned for real-time performance. For real-time systems the entire implementation should be focused on performance and the use of existing open-source software may not be suitable. Also, other transaction models (e.g. hierarchic [21]) than a centralized transaction manager receiving join-messages from all participants should be invesitgated.

For a real world application the advantage of increased availability must be weighed against the cost of replication. If the system cannot tolerate the downtime caused by a restart of a machine, replication should be used. On the other hand, if the increased response time cannot be tolerated, but a few minutes of unavailability once in a while can be, replication should not be used.

The system developed in this paper is a prototype where several shortcuts have been made to get a working system for basic testing. To be of any practical use, it must be able to restart crashed replicas, initiate new ones and update the new replicas with the current state. The Jgroup/ARM system has support for automatically performing these actions, but it has not yet been implemented in this prototype. Also, the system must be able to handle all failure scenarios during 2PC to be able to terminate all transaction in the presence of failures.

## References


1. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)
2. Helal, A.A., Bhargava, B.K., Heddaya, A.A.: Replication Techniques in Distributed Systems. Kluwer Academic Publishers (1996)
3. Budhiraja, N., Marzullo, K., Schneider, F.B., Toueg, S.: Distributed systems. In Mullender, S., ed.: Distributed Systems. ACM Press. second edn. Addison-Wesley (1993) 199–216
4. Schneider, F.B. In: Replication management using the state machine approach. ACM Press/Addison-Wesley Publishing Co. (1993) 169–197
5. Pleisch, S., Kupšys, A., Schiper, A.: Preventing orphan requests in the context of replicated invocation. In: Proceedings of the 22nd International Symposium on Reliable Distributed Systems, Florence, Italy, IEEE (2003) 119 – 128
6. Poledna, S.: Replica determinism in distributed real-time systems: A brief survey. Research Report 6/1993, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria (1993)
7. Gray, J.: Notes on data base operating systems. In: Operating Systems, An Advanced Course, London, UK, Springer-Verlag (1978) 393–481
8. Reddy, P.K., Kitsuregawa, M.: Reducing the blocking in two-phase commit protocol employing backup sites. In: Proc. of CoopIS. (1998)

9. Felber, P., Narasimhan, P.: Reconciling replication and transactions for the end-to-end reliability of CORBA applications. In: On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002, Springer-Verlag (2002) 737–754
10. Frølund, S., Guerraoui, R.: Implementing e-transactions with asynchronous replication. Dependable Systems and Networks (2000) 449–458
11. Montresor, A.: System Support for Programming Object-Oriented Dependable Application in Partitionable Systems. PhD thesis, University of Bologna, Italy (2000) Technical Report UBLCS-2000-10.
12. Sun Microsystems Inc.: Jini Technology Core Platform Specifications. 2.1 edn. (2005)
13. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems (3rd ed.): Concepts and Design. Addison-Wesley Longman Publishing Co., Inc. (2001)
14. Object Management Group: Transaction Service Specification. (2003) OMG Technical Committee Document formal/03-09-02.
15. Object Managment Group: Fault Tolerant CORBA. (2004) OMG Technical Committee Document formal/04-03-21.
16. Little, M.C., Shrivastava, S.K.: Integrating group communication with transactions for implementing persistent replicated objects. j-LECT-NOTES-COMP-SCI **1752** (2000) 238–253
17. Dekel, E., Goft, G.: ITRA: Inter-tier relationship architecture for end-to-end QoS (2001)
18. Frølund, S., Guerraoui, R.: Transactional exactly-once. Technical report, Hewlett-Packard Laboratories (1999)
19. Pleisch, S., Kupšys, A., Schiper, A.: Replicated invocations. Technical report, Swiss Federal Institute of Technology (EPFL) (2003)
20. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison-Wesley Longman Publishing Co., Inc. (1986)
21. Mohan, C., Lindsay, B., Obermarck, R.: Transaction management in the R* distributed database management system. ACM Trans. Database Syst. **11** (1986) 378–396
22. Arnold, K., Scheifler, R., Waldo, J., O'Sullivan, B., Wollrath, A.: The Jini Specification. second edn. Addison-Wesley Longman Publishing Co., Inc. (2001)
23. Montresor, A., Davioli, R., Babaoğlu, Ö.: Jgroup: Enhancing Jini with group communication. In: Proceedings of the ICDCS Workshop on Applied Reliable Group Communication. (2001)
24. Moland, R.: Replicated transactions in Jini. Master's thesis, University of Stavanger (2004)
25. Hvasshovd, S.O., Torbjørnsen, Ø., Bratsberg, S.E., Holager, P.: The ClustRa telecom database: High availability, high throughput, and real-time response. In: VLDB. (1995) 469–477