Preface

The computer world of today becomes increasingly distributed causing applications to rely on remote servers. Thus, the probability of a partial failure of the system increases. This might cause unavailability of a service and inconsistencies can arise. Depending on the service and system affected, this might cause anything from a slight inconvenience for a single user to bankrupting a large company. Therefore, such failures should be masked to keep the system available and consistent.

Transactions and replication are both techniques to make a system fault tolerant. The first is designed to protect the data from inconsistencies, while the latter protects the processing, ensuring availability. A complete integration of the two yields a consistent and available system which masks failures.

Operations like a random number generator cause the execution to be non-deterministic. These operations cannot be redone just by executing the same code again. Thus, special handling is needed when servers with non-deterministic execution fails.

This document presents a framework for the integration of transactions and replication, while supporting non-deterministic execution. An implementation of the framework on top of Jini and Jgroup/ARM is outlined and test results are presented and discussed.

The thesis is submitted to the Department of Computer and Information Science (IDI) at the University of Science and Technology (NTNU) as partial fulfillment of the degree "Sivilingeniør" (MSc). The work has been carried out at the Database Systems Group.

Acknowledgements

A lot of people have assisted me during my work on the thesis. Thanks goes to my supervisor Svein-Olaf Hvasshovd for his valuable comments and critical questions on the transactions-technical parts of my work.

Jørgen Løland has been of invaluable help in proofreading and input for typesetting and graph design. Thanks for sustaining my questions every ten minutes for the last month.

Thanks to Donald E. Knuth and Leslie Lamport for writing TEXand LATEX, respectively. Their typesetting tool makes it a lot easier to make a document look scientific.

Other people who have supported me and given comments include: Njål Karevoll, Runar Johannes Solberg, Knut Karevoll, Dagny Kolltveit, Øyvind

Vestavik, Gisle Grimen, Jeanine Lilleng, Jon Olav Hauglid and probably some others.

Abstract

This thesis presents a framework of a passively replicated transaction manager. By integrating transactions and replication, two well known fault tolerance techniques, the framework provides high availability for transactional systems and better support for non-deterministic execution for replicated systems. A prototype Java implementation of the framework, based on Jgroup/ARM and Jini, has been developed and performance tests have been executed. The results indicate that the response time for a simple credit-debit transaction heavily depends on the degree of replication for both servers and the transaction manager. E.g. a system with two replicas of the transaction manager and the servers quadruples the response time compared to the nonreplicated case. Thus, the performance penalty of replication should be weighed against the increased availability on a per application basis.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Background and Context

The topic of this thesis is fault tolerance and high availability for distributed systems. The focus is on replication of the transaction manager while supporting non-deterministic execution. The motivation for choosing this topic and the methods used to approach it and the organization of the thesis is presented in the following sections.

1.1 Motivation

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

Unavailability can cause great damage to anything from a single user to a company and possibly millions of users of the company's services. If, for instance, a telecommunication or power grid system becomes unavailable, the costs could bankrupt the company. Applications that are less critical for the society could also be harmed by unavailability. If a website is unavailable, potential and existing customers will probably use the competitor's site instead. This may lead to loss of revenue.

Inconsistency in a system can also lead to major problems. For instance, an inconsistency in a banking system could lead to an incorrect withdrawal of an arbitrary amount of money, and a webshop company might send out the wrong package, maybe to the wrong person, or the customer might not be charged for it.

Two concepts to deal with unavailability and inconsistency, called transactions [GR93] and replication [HBH96], have been proposed in the literature.

A (correctly written) transaction obeys the ACID-properties (atomicity, consistency, isolation and durability), and therefore guarantees a consistent execution of the operations contained in it. Atomicity, often called the all-or-nothing property because it ensures that either all or none of the operations contained in the transaction is performed, is enforced by a transaction manager (TM) using an atomic commitment protocol (normally 2PC). Hence, transactions can be used to avoid jeopardizing the consistency of the system by aborting (rollback)

the changes made by a single transaction.

Replication provides availability by making sure there are more than one server to provide any given service. As such, replication can be used to ensure the availability of the system by rolling forward the execution to an available server.

Fault tolerance is needed both for protecting the data and for protecting the processing. Transactions are often used as a way to protect data, while replication is used for the protection of processing. Together, these two provide availability and consistency for a system. If these two mechanisms are integrated rather than being kept as separate concepts, two additional properties can be provided as well[FN02]: First, a higher level of consistency for replicated systems is achieved due to better support for non-deterministic execution and, second, a higher level of availability and better failure masking for transaction processing systems.

In a fault tolerant system, no single point of failure should exist. Any component that is a single point of failure will, in case it fails, render the entire system unavailable. Therefore, not only the application servers, but all system components, particularly the transaction manager, need to be replicated to provide system-wide availability.

This thesis will describe the changes needed to make a transaction manager fault-tolerant by replication.

1.2 Approach

The topic is approached by designing and implementing a framework of a passively replicated transaction manager. A full-fledged transaction manager is a complex system component and implementing replication is not straightforward. In addition, empirically test have shown that reuse of software can result in shorter development time, better productivity and less problems [Moh04]. Therefore, the implementational part of this thesis is based on existing toolkits for replication (Jgroup/ARM) and transactions (Jini)

Jgroup/ARM[1][Mon00] is an object group system written entirely in Java [Sun05]. It was developed to integrate group technology with distributed objects. An object group is a replicated object. Clients can invoke the object group as if it was just a single object. The replication is transparent to the invoker. Jgroup is already integrated with Jini[2] [ASW+01] by a replicated registry service [MDB01] and an actively[3] replicated transaction manager [Mol04].

By using the existing libraries and source code for Jgroup and Jini, the implementation part is reduced to a manageable problem within the given timeframe.

The approach used in this thesis is first to present the central concepts in Chapter 2. Then a state-of-the-art survey, which establishes a foundation for this work, is given in Chapter 3. The environment for the implementational part of the thesis is presented in Chapter 4, and an introduction to replicated invocations and the problems they cause is given in Chapter 5. How to combine transactions and replication in the context of Jgroup is discussed in Chapter 6. Chapter 7 describes the implementation of the passive transaction manager,

[1] Jgroup/ARM is licensed under GNU Lesser General Public License (LGPL) [GNU]
[2] Jini is licensed under Apache License, Version 2.0 (ALv2) [ALv]
[3] The difference between passive and active replication is explained in Section 2.2.

while Chapter 8 presents some test cases with results executed on the implemented system. The results of the tests are discussed in Chapter 9. Finally, Chapter 10 concludes the thesis and gives direction for further research.

Chapter 2

Concepts

This chapter presents the most important concepts regarding software-based fault-tolerance. Introductions to transactions, replication, and liveness and safety are given here.

2.1 Transactions

Transactions were developed in the context of database applications to provide a structured way to deal with failures. A transaction is a collection of operations or actions with the ACID properties [GR93, BHG86]:

- **Atomicity** - After the completion of a transaction all operations of the transaction must have executed successfully or none of them must appear to have executed. This is also known as the all-or-nothing property of transactions or the everyone-or-none property for the distributed systems.
- **Consistency** - The sum of the operations in a transaction must transform the applications from one consistent state to another consistent state.
- **Isolation** - The effects of at transaction must not be visible to other transactions before it has committed.
- **Durability** - The effects of a successfully completed transaction must be permanent once the transaction has committed.

If a failure happens while a transaction is executing, the transaction is aborted, and all of the state changes caused by the transaction are undone. This rollback of the transaction is the most common implementation of the atomicity property, and thus transactions are a safety-focused mechanism: If a failure occurs the state is rolled back to a previously consistent state.

A *transaction manager* is responsible for coordinating a protocol to ensure the atomic commitment of a distributed transaction. The protocol is usually the two-phase commit protocol (2PC) [GR93, BHG86], but other protocols also exist, e.g. 3PC [Ske81], FCWFA and FAWFC [DGP04], LLNBCS [JPPMAA01], and ACP-SB and ACP-UTRB(1-3) [BT93].

Generally, an atomic commitment protocol gathers votes from all participants of a transaction and then aborts or commits the entire transaction. Since

2PC is a well documented protocol [GR93, BHG86] and there are three versions of it [MLO86]; presumed abort, presumed commit and presumed nothing, only a short outline of the one used in the implementation, presumed abort (2PC-PA), is given here.

As the name indicates, 2PC-PA uses two phases. The first phase is the voting phase. First, the coordinator requests and receives the votes from the participants joined in the transaction. Each participant persistently saves its vote before replying. If all the votes are positive, the transaction is decided to commit, otherwise the decision is to abort. A decision to commit is persistently saved, while a decision to abort can be forgotten. The second phase sends the outcome of the transaction to the participants, which persistently saves it if it is commit, then acknowledges it back to the transaction manager.

2.2 Replication

In a distributed environment any process may fail at any given time. A single point of failure is a component that can jeopardize the availability of the system if it fails. Examples of such components include routers, communication lines, CPUs, disks, etc. If, for instance, a process executing an important service fails, the entire system might be rendered unavailable. By introducing redundancy by replicating the components, the system can stay operational even in the case of a process failure. In this way, all single points of failure can be eliminated. In this section and in the rest of this thesis, only replication of processing is discussed.

Replication of servers may seem straightforward, but to ensure both consistency and availability of the system, special care must be taken. For instance, a protocol to manage the replicated server group is needed. Also, the replicated invocation problem must be handled. A *replicated invocation* is an invocation from a replicated server to another, possibly replicated, server. Depending on the kind of replication (see below) used, both the group management facilities needed and the replicated invocation problem change.

There are two main types of replication; *active* [Sch93] and *passive* [BMST93]. For an actively replicated server group an invocation from the client (group) is multicast to all of the replicas, and each of them perform the same task and reply to the client. This is shown in Figure 2.1. Group management and communication is required to ensure that all of the servers see the same set of messages and thus stay consistent. For instance, heartbeat messages are used to create a view of operational servers.

A passively replicated server group has a leader (primary server) that receives and processes invocations. The primary updates the backups periodically or at certain events, for instance after each completed invocation as illustrated in Figure 2.2. The process of updating backups is called *checkpointing*. The backups process these updates and in the case of a primary failure, the backups elect a new primary to take over the processing. This election is carried out by a group management protocol.

Other types of replication, e.g. semi-active [VBB+91] and semi-passive [DSS98], also exist. Défago and Schiper [DS02] gives a more thorough presentation of these replication techniques.

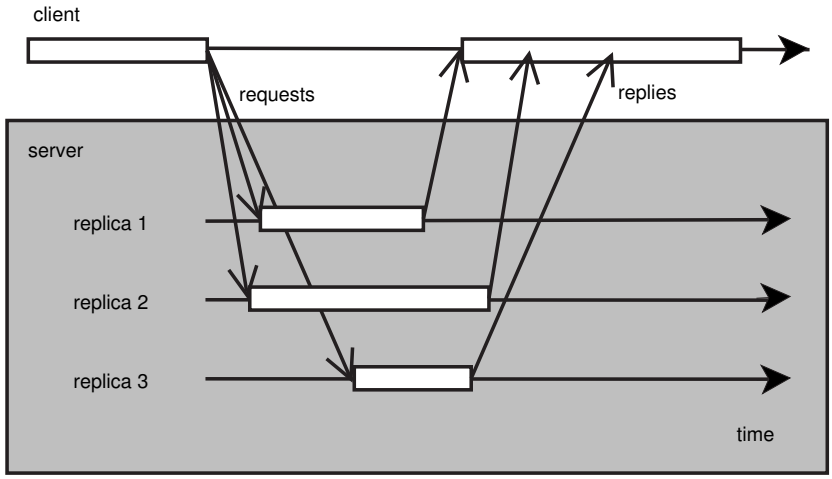


Figure 2.1: Active replication

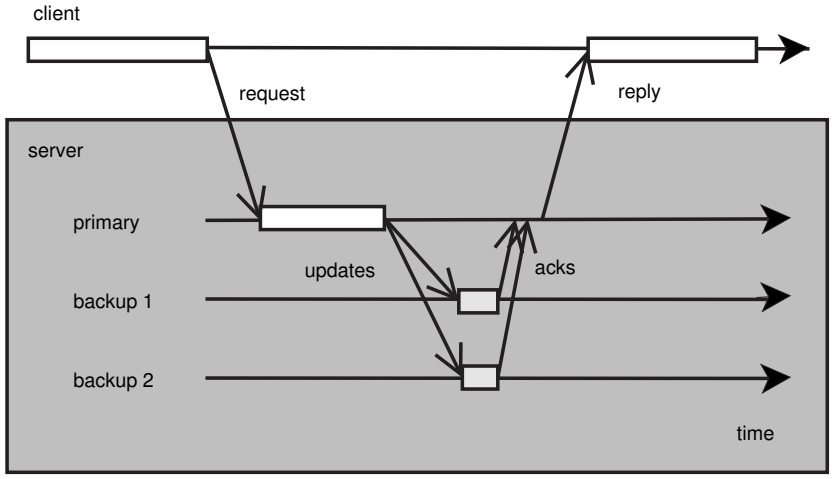


Figure 2.2: Passive replication

2.2.1 Replica Determinism

Some replication techniques, like active replication, require all replicas to behave identical. If, for instance, there are two replicas of a server group that give diverging replies to a client, it has no way to know which result is the correct one. Also, if the internal state-changes of the server replicas differ, it will lead to inconsistencies that may later spread to other parts of the system. Therefore, all *correct* replicas must agree to the same result. This can be done by making the execution of the replicas *deterministic*.

Schneider [Sch90] defines replica determinism as follows: "A replica group is deterministic if, in the absence of faults, given the same initial state for each replica and the same set of input messages, each replica in the group produces the same ordered set of output messages." Poledna [Pol93] argues that this definition is too restrictive, as it is does not cover all replication techniques and it does not capture the timing requirements of real-time systems. For this thesis, however, the above definition by Schneider suffices.

There exist many sources that can render the execution non-deterministic. The most obvious one is a mathematical random-function, but more subtle sources are also at work. A timeout is one example: Say, for instance, that a server group waits for a reply from another server and the reply is received close to the expiration of the timeout. Some servers might decide that it came just in time, while others might decide it came too late. Even with global coordination of the timeout, their respective decisions can diverge because of tiny differences in the processing speed or network delay. Other sources of non-determinism include inconsistent inputs from analog sensors, multithreading, inconsistent order of requests and non-deterministic programming language specific constructs [Pol93].

2.3 Liveness and Safety

Both liveness and safety are properties needed to make a system available and consistent. The first property causes a system to eventually do something good (e.g. the eventual completion of an operation and the delivery of a reply), while the latter causes a system to do nothing wrong (i.e. not leaving it in an inconsistent state) [AS85].

Transactions are a rollback mechanism. Faults are tolerated by undoing the operations that lead to the failure, keeping the data consistent. On the other hand, replication is a roll-forward mechanism. Faults are tolerated by rolling forward and continuing the process execution on another replica. Thus, transactions and replication are safety and liveness focused concepts, respectively, [FN02] and they are both important aspects of a fault-tolerant system.

Chapter 3

State of the Art Survey

Many projects that deals with replication and transactions, and some that deals with the combination of the two exists. In this chapter, the most influential object group projects and integration techniques are presented.

3.1 Object Groups

Over the last decades, many projects have introduced redundancy to achieve fault tolerance for distributed applications. Most object groups research efforts derive from Java RMI [Sun99] and the CORBA specification [OMG04a], and are based on the group communication paradigm [Bir93]. First, an introduction to CORBA and CORBA related object groups are presented, followed by projects dealing with JAVA RMI object groups.

3.1.1 CORBA Based Object Groups

The Common Object Request Broker Architecture (`CORBA`) is a specification of a distributed system [TS01, chapter 9]. It has been made by the Object Management Group (OMG)[1] and was designed to overcome interoperability problems regarding the use of different operating systems and applications in a single system. The current release is version 3.

A basic overview of the architecture of `CORBA` is shown in Figure 3.1. The core of this model is the *Object Request Broker* (*ORB*). It enables the communication between clients and objects, and makes the distributed and heterogeneous nature of the system transparent to the application programmers and users. The Common Object Services are the most widely used services and include services for naming, concurrency control, events, life cycling, recovery, persistence, security, etc.

Horizontal facilities are high-level services not related to any specific application domain. Examples include user interfaces and information as well as system and task management. The vertical facilities are high-level services that are targeted towards specific application domains such as banking and manufacturing.

[1] www.omg.org

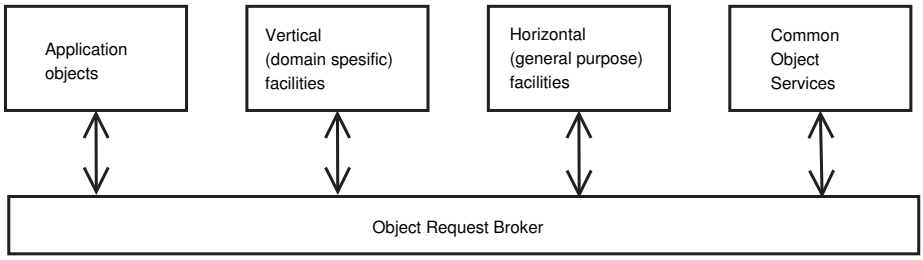


Figure 3.1: Overview of CORBA [TS01]

CORBA related projects include Eternal [Nar99], Electra [Maf95], DOORS [CHY+98, GNSY00],, Newtop [MSEL99], OGS [FGS98], AQuA [RBC+03] and the adopted OMG standard FT-CORBA [OMG04b]. Meling et al. [MMBH02] classifies the projects into three categories relative to the placement of the object group support; the *integration approach*, the *interception approach* and the *service approach*. In the first approach the support for object groups is integrated into the ORB. The ORB is then responsible for multicasting an invocation to the object group. Electra followed this approach. Eternal is based on the second approach, which intercepts requests and replies at the client and server by the OS. The messages are then sent to the correct replicas. The last approach implements group communication as a separate horizontal service, and can thus be used in any CORBA implementation. OGS, DOORS, Newtop and FT-CORBA are examples of this approach.

3.1.2 Java RMI Based Object Groups

There are several Java RMI based projects that offer object groups. Filterfresh [BCH+98] supports logical object groups that allow the continuation of requests on a backup replica. However, it lacks client to server invocation with multicast semantics and group internal use of RMI. Another toolkit for reliable group communication is JavaGroups [Ban98]. It supports multicast, but its implementation does not provide transparent RMI for the client. Aroma [Nar01] is the Java RMI version of Eternal, thus it is dependent on low-level OS-specific interception and group communication mechanisms. Also, there is an extension to NinjaRMI [LMW] that provides basic one-to-many RMI, but nothing more.

Jgroup/ARM [Mon00, MMBH02] is implemented entirely in Java and does not rely on any underlying toolkits or special operating systems. All communication is done through group method invocations using Java RMI, and it is the only system that is fully partition-aware and provides automatic management of replicas [MH01]. The implementation presented in Chapter 7 in this thesis is based on Jgroup.

3.2 Integration of Replication and Transactions

Replication and transactions have historically been two separate techniques for achieving fault-tolerance. FT-CORBA [OMG04b] is an example of system treating them separately. A distributed transaction system can profit from using

process groups [LS00], in particular, stronger consistency and higher availability can be achieved if the techniques are integrated [FN02]. This is discussed in more detail in Chapter 6.

Schiper and Raynal [SR96] presents an approach where transactions are implemented with group communication primitives. All operations of a transaction are put into one message, which is sent to all process groups participating in the transaction. Because of the total order and atomic delivery [TS01, pp. 389–391] provided by the group communication system, the transaction will be atomic and serializable, and the replicated nature of the process group will ensure durability. However, this approach assumes that process groups will always be able to complete the operations contained in the message. This is an unreasonable constraint since a process group might be unable to execute all operations because of internal constraints. For example, a banking account may not be overdrawn. In a normal transaction processing system this will cause the process group to vote abort, but since no abort mechanism exists it may result in an inconsistent system state.

Little and Shrivastava [LS00] study 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 failover and active replication.

Other systems that support transactions in a replicated environment also exist. GroupTransactions [PMJPA01] allows transactional servers to be process groups, and thus provides an integration of the concepts. `Circus` [Coo85] extends the replicated environment with support for multi-threaded servers through a novel commit and synchronization protocol. Zhao et al. [ZMMS02] replicate the transaction coordinators, which leads to a non-blocking 2PC protocol, a highly desirable property since it ensures liveness. All of these systems are actively replicated which requires the servers to be deterministic.

Systems that support non-deterministic execution must be able to control its effects. ITRA [DG01] is an approach that handles them by replicating the result of each non-deterministic operation to the backups. Though ITRA claims to support transactions, the exact integration is not presented. Frølund and Guerraoui [FG99] present 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. [PKS03a, PKS03b] 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 undo information to the backups before invoking a server, orphan subtransactions can be terminated in the pessimistic case, and compensated in the optimistic case. A CORBA related approach [FN02] avoids the need for undo information by using a centralized transaction manager and nested transactions. If a parent transaction cannot be completed, the transaction manager aborts all subtransactions. These systems are presented in more detail in Section 5.2.2.

The concept of *process pairs* [GR93] was developed in the context of databases and combined with transactions to achieve highly available and highly reliable processes. A process pair is the equivalent of a passively replicated object group. A primary executes requests from the clients and sends state updates to the backups. Heartbeat messages are used to detect a failed primary. When that

happens, a protocol to agree on a new primary is executed. This approach assumes deterministic execution.

Part II

Framework

Chapter 4

Environment

This chapter presents the layout of the implementation environment. First, an overview of the Jini architecture and its most central services related to this thesis and Jini Extensible Remote Interface (JERI) is provided. The architecture and the central concepts of the Jgroup/ARM system is then presented

4.1 Overview of Jini

Jini technology [ASW+01] is an infrastructure that allows services and devices to cooperate in a distributed environment. It provides a way for clients to find and access services as objects.

Each service publishes a Java object which implements a service. When a client finds the object of an implemented service API that it wishes to use, it downloads any code necessary to communicate with the service. Thus, the interaction can be provided by any distributed network technology such as Java RMI [Sun99], CORBA [OMG04a] and SOAP [Con03].

The Jini architecture can be split into three categories; infrastructure, programming model and services. Figure 4.1 shows this segmentation. The infrastructure enables the distributed system to be built, while the services are

	Infrastructure	**Programming Model**	**Services**
Base Java	Java VM RMI Java Security	Java APIs JavaBeans™ ...	JNDI Enterprise Beans JTS ...
Java + Jini	Discovery/Join Distributed Security Lookup	Leasing Transactions Events	Printing Transaction Manager JavaSpaces™Service

Figure 4.1: Jini architecture segmentation [Inc03a]

the top-level entities that constitute the system. The programming model in-between is the interfaces that enable reliable services to be built.

To facilitate different needs for fault tolerance and persistency, all services in Jini can be run in three different failure modes, set for each service in the Jini configuration files:

- **Transient** - The state of the service is only kept in main memory. Thus, if it fails, the state of the system is lost.
- **Persistent, non-activatable** - The state of the service is saved to persistent storage, but after a crash failure, it needs to be manually restarted.
- **Persistent, activatable** - The state of the service is saved to persistent storage, and after a crash failure, it is automatically restarted with the state before the crash.

The core services, which are relevant for this thesis, will be described in the subsequent sections.

4.1.1 The Lookup Service

Services in Jini are dynamically registered in a lookup service [Inc03b], called *Reggie*. Reggie runs a *discovery* protocol to find services that has joined the network. When a service is found, it is joined in the lookup service by a *join* protocol. Each service is registered with a proxy and possibly some attributes describing the properties of the service. Whenever a client asks Reggie for a certain service, the proxy is returned to the client if found. The client can then access the service through that proxy.

Each service has a 128-bit identifier that uniquely identifies it in the system. It can be generated dynamically when the service is registered with Reggie, or statically by the vendor at deployment.

4.1.2 Jini Transaction Service

The interfaces that constitute the programming model include interfaces for transactions. Jini comes with an implemented transaction service named Mahalo [Inc03b]. It implements the methods necessary to create transactions, to join participants and to terminate transactions. The transaction outcome is determined by a two-phase commit protocol.

Unlike traditional transaction processing (TP) systems, Mahalo has no TP monitor [GR93] that controls the transaction semantics. It is left to each participant of the transaction to implement the semantics in the best possible way for that kind of participant. This means that the association of a request with a transaction is done by the participants, as well as all authentication, authorization, scheduling and recovery. This allows the all-or-nothing property to be expanded beyond the classical database system.

When a client needs a new transaction, the `TransactionFactory` object in the client invokes the transaction manager to get a unique transaction identifier. The identifier is used to create a `Transaction` object, as shown in Figure 4.2. Other services join the transaction when they are invoked by the client with the `Transaction` object as an argument. They become *participants* of the transaction.

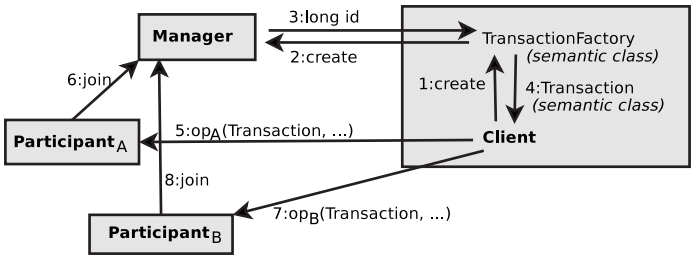


Figure 4.2: Transaction creation and use [Inc03b]

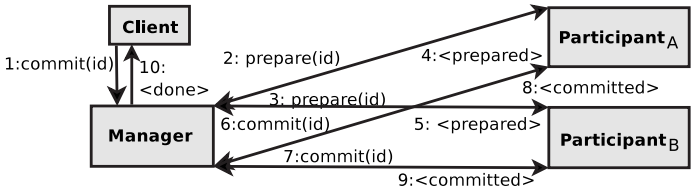


Figure 4.3: Successful transaction termination

When all participants have completed their requests, it is up to the client to initiate the termination protocol. The protocol used in Jini 2.0 is the two phase commit protocol (2PC). Figure 4.3 shows a successful termination of a transaction using 2PC. The client sends a commit request containing the transaction identifier to the manager. The manager then starts the voting phase, asking each participant if they are able to commit the transaction. This is shown as messages 2 and 3 in the figure. Messages 4 and 5 are the affirmative replies from both participants. Since all participants have voted yes, the manager stores the decision to commit persistently, and tells all participants to commit. After completion, the manager reports the outcome of the transaction to the client. Özsu and Valduriez [ÖV99] and Gray and Reuter [GR93] presents more details, including failure handling, regarding the termination of transactions using 2PC.

Jini also supports nested transactions, but this feature has not being used in the thesis, and will thus not be discussed any further.

Transaction Manager

An implementation of a Jini transaction manager must implement the Jini `TransactionManager` interface and has three main responsibilities:

- It creates the transaction identifier along with a lease-time, which the `TransactionFactory` uses to create the `Transaction` semantic object.
- It keeps track of the participants that have joined a transaction.
- It is responsible for terminating the transaction, i.e. to collect the votes from the participants and decide the outcome as a part of the two-phase protocol.

Transaction Participants

A transaction participant is a service (usually an application program) that must support the ACID properties of transactions. Consequently, it must be able to rollback any changes the operations of the transaction might have performed. When an operation with a new semantic `Transaction` object arrives, it must join the transaction by calling the `join` method of the transaction manager. All transaction participants in Jini must implement the `TransactionParticipant` interface, and thus support the three methods the transaction manager may invoke on them as a part of 2PC: `prepare`, `abort` and `commit`.

Transaction Clients

A client for a Jini transaction initiates the transaction by telling the `TransactionFactory` to create a new transaction. As noted in Section 2.1, a transaction is used to group a number of operations together to form a unit, to ensure that either all or none of the operations are performed. The client invokes these operations on Jini services. To keep track of the semantics of the transaction the `Transaction` semantics object is sent with each invocation that is a part of the transaction.

The client may at any time invoke the `abort` method of the transaction manager to rollback the changes made by the transaction so far. Normally, if all goes well, the client will initiate 2PC by invoking the `commit` method when all operations have been performed. Clients may also join the transaction as a participant and thus take part of 2PC. A client can be either a user application or a service that makes use of other services.

4.1.3 Jini Extensible Remote Invocation

Java RMI [Sun99] allows remote method invocation to be syntactically equal to invoking a local method, while acknowledging the semantic differences. However, it suffers from two major shortcomings [Som03]:

No security mechanisms. As RMI uses mobile code, and mobile code may be malicious, it is suited for trusted local area networks (LAN) only. To extend it to wide area networks (WAN) and over the Internet, security mechanisms are required.

No easy configuration. There is no way to configure a Java RMI implementation. A configuration API is needed.

Jini Extensible Remote Invocation (JERI) was made by Sun developers to cover RMI's shortcomings. By using JERI the application developer can choose any Java RMI implementation, e.g. SOAP, IIOP, JERI or JRMP. This allows for better interoperability with other programming languages. It also eliminates the need for compile time generation of proxies and has better distributed garbage collection [New04].

JERI implements a new layerstack that facilitates the flexibility of the protocol, as shown in Figure 4.4. The invocation layer marshals and unmarshals the remote invocation's objects and parameters, while the object identification layer tracks and manages several aspects, such as distributed garbage collection,

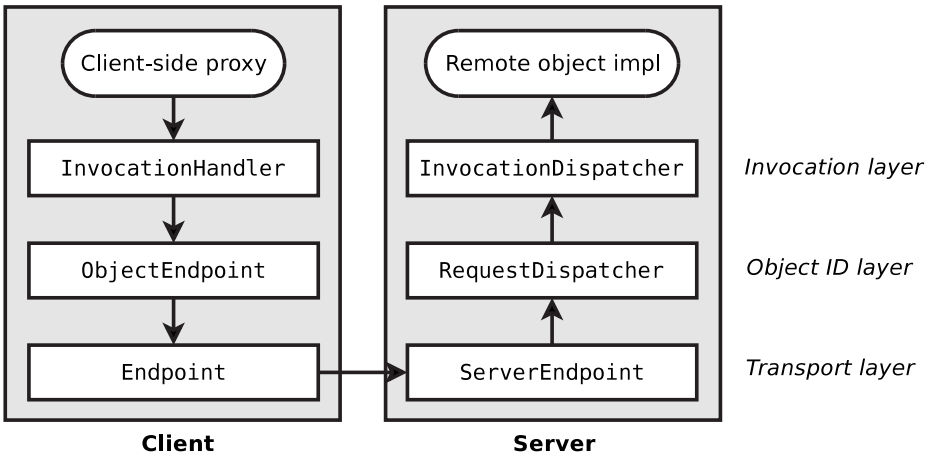


Figure 4.4: The stack of layers used in JERI [Som03]

of exported objects. Finally, the transport layer is responsible for the protocol used for transporting the invocation (http, tcp, ssl, etc.).

4.2 Overview of Jgroup/ARM

Fault tolerance and high availability for object-oriented systems usually works by replicating an object into an object group and masking the replication. The system distributes invocations on the object group according to the replication scheme (passive, active, etc.). Jgroup [Mon00] is based on the object group paradigm and is implemented entirely in Java. It executes client invocations on the object group through an `External Group Method Invocation` (EGMI) module, thus making the replication transparent to the client.

Jgroup extends the object group paradigm with the following main components:

- A Partition-aware Group Management Service (PGMS)
- A State Merging Service (SMS)
- A Group Method Invocation Service (GMIS)

The PGMS keeps track of the group membership of servers. It provides a consistent view of the members of the group to each of its members by informing them of changes in the views. The changes may be due to failures in the system, installation of new members or removal of existing ones. As indicated by the name, the PGMS supports partitions and thus multiple concurrent views.

The partitioned server state can be recombined by the SMS. It consists of two methods, one for retrieving the state, and one for merging the received state with the existing one. As such, the SMS provides a way for servers to install a common state after the partitioning or initializing of a server.

Reliable communication is provided by the GMIS. It is the responsibility of GMIS to execute methods on the servers according to the replication policy of

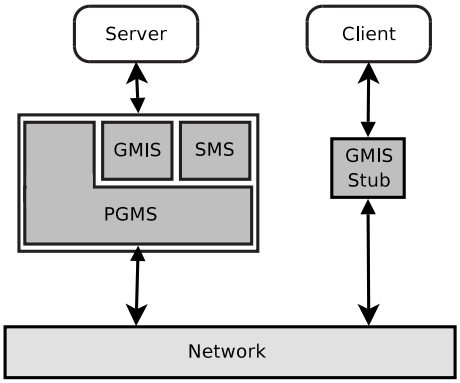


Figure 4.5: Overview of the core services in Jgroup [Mon00]

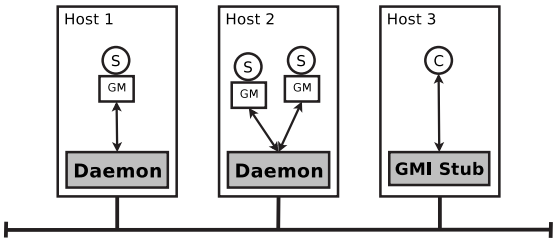


Figure 4.6: The Jgroup architecture [Mon00]

each method. Clients can only see the client-side group proxy of the remote server, and cannot distinguish a GMI interaction from a local invocation. The GMIS treats invocations between servers different from invocations between clients and servers. This is elaborated on in Section 4.2.5

Architecturally, Jgroup/ARM consists of two main components; the group manager and the daemon (see Figure 4.6). These are presented in the following sections, along with the autonomous replication management framework, the group enabled registry and group method invocations.

4.2.1 The Group Manager

The interaction between the Jgroup/ARM core services and the application objects is controlled by the `GroupManager` [Mon00]. It is based on a stack of layers, each providing a service to the application. As can be seen in Figure 4.6, each server, S, has a `GroupManager` associated with it that keeps control over the services needed by the group.

4.2.2 The Daemon

The `Daemon` provides the membership and the multicast services. As indicated in Figure 4.6, more than one `GroupManager` can connect to the same `Daemon` to make use of its services. Interactions between the `GroupManager` and the `Daemon` are executed as Java RMI invocations.

4.2.3 The Autonomous Replication Framework

Jgroup includes an Autonomous Replication Framework (ARM), which is responsible for deploying and operating dependable services within a predefined environment [MH01, MMBH02]. The environment is the nodes where replicas of the services can be hosted. It distributes and replicates the servers according to rules specified by distribution and replication policies. For instance, the level of redundancy offered by a service can be specified. If the current replica count drops below this level, ARM automatically initializes a new replica of the server within the environment.

4.2.4 Greg: Jini Group-Enabled Lookup Service

As mentioned in Section 4.1.1, Jini comes with a lookup service called Reggie. However, it does not handle replicated services. Montresor, Davoli and Babaoğlu [MDB01] extended Reggie to support object groups. The group-enabled service is called *Greg*.

Greg uses the same protocol as Reggie to discover services and register the proxy. But instead of having one static proxy per service, Greg uses a dynamic *group proxy* that includes all replicas of the service. When a new service replica is found, the existing group proxy is not overwritten, just modified. Thus, when a client does a lookup on the service, a group proxy containing information about all replicas is returned. The group proxy is then used to contact the service according to the given invocation semantics.

4.2.5 External and Internal Group Method Invocations

All communication in Jgroup/ARM is provided by *group method invocations*. This allows for a uniform treatment of communication. Jgroup/ARM distinguishes two different types of group method invocations. A client communicates with the server group through the *External Group Method Invocation* (EGMI) facility. This makes the server replication transparent to the client. On the server side an *Internal Group Method Invocation* (IGMI) facility exists to aid the coordination of the replicas' global actions.

The rest of this section describes IGMI and EGMI in more detail.

Internal Group Method Invocation

An IGMI returns an array of results, one from each server in the group. In case of a failure to complete it returns one of the exceptions associated with the method invocation. All IGMI methods for a group are declared in an interface. The IGMI service is retrieved by a call to the `GroupManager` and can then be accessed like traditional remote method invocation.

External Group Method Invocation

The JERI EGMI layer supports EGMI. It is a modified version of Jini ERI, which is presented in Section 4.1.3. It is extended to support group communication to be able to perform group invocations. As for IGMI, the methods that can be performed on the group must be declared in an interface An EGMI service is retrieved by a call to the registry like explained in Section 4.1.1, but a

group enabled registry as the one presented in 4.2.4 is needed to support group invocations.

Currently, there are four different invocation semantics for EGMI. These are:

- *Multicast.* An invocation on an object group is invoked on *all* replicas in the group.
- *Leadercast.* An invocation on an object group is only invoked on the *leader* replica of the group.
- *Anycast.* An invocation on an object group is invoked on a random replica of the group.
- *Atomic.* An invocation on an object group is invoked with total-ordering [BJ87].

The semantics apply not on a per object basis, but on a per method basis. This allows for greater flexibility as various methods may need different semantics. For instance, a read-one-write-all protocol [TS01] could easily be implemented by using anycast for read operations and multicast for write operations.

The type of semantics for each method is specified by using annotations[1]. To specify that a method should be invoked with leadercast, multicast or atomic semantics the method declaration is prefixed with `@Leadercast`, `@Multicast` or `@Atomic`, respectively. If none are given, anycast is used.

For a client application there is no difference between an EGMI and a traditional Java RMI. This provides complete *transparency* of server replication.

[1] http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

Chapter 5

Replicated Invocations

This chapter starts by presenting replicated invocations and the problems they pose to the consistency of a replicated system. General solutions to these problems are then discussed

The problem description is heavily based on a state of the art report on highly available applications by the author [KH04].

5.1 Problem Description

A *replicated invocation*, also called *chained* invocation, happens when a replicated server invokes another server. The invoked server can be replicated, but it does not have to be. The resulting state-changes and result of such an invocation must be identical to an invocation not involving replicated servers. As explained in the following sections, the problem to solve depends on whether the invoking replicated server is deterministic or not.

5.1.1 The Duplicate Invocation Problem

Consider an actively replicated server A and a server B that may or may not be replicated. Whenever A invokes an operation on B, each of A's replicas sends a request to B. Since an actively replicated server must be deterministic (see Section 2.2.1), all of these requests are identical. If the request is idempotent, meaning that multiple invocations have the same end-effect as a single invocation, there is no problem. Since requests usually are non-idempotent, however, a mechanism to detect and handle duplicate requests is needed. This is called the *duplicate invocation problem*.

Figure 5.1 shows an example of the duplicate invocation problem for two actively replicated servers A and B. The client sends a request to each replica of server A, a_0 and a_1. a_0 and a_1 both process the request and issue a new request to the replicas of server B, b_0 and b_1. b_0 and b_1 will both receive two separate, but identical requests. The duplicate requests that must be handled are shown as light-grey arrows. As can be seen, there are also duplicate replies that must be taken care of. Fortunately, these can be dealt with by exactly the same mechanisms as the duplicate requests.

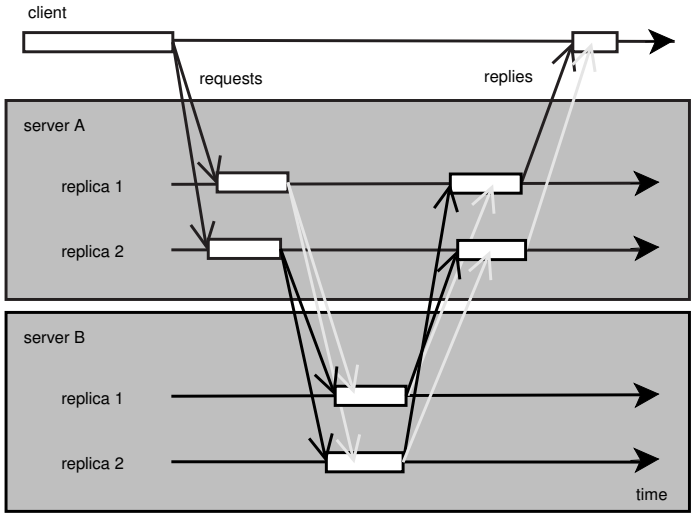


Figure 5.1: An example of a duplicate invocation

Passive, semi-passive and semi-active replication support both deterministic and non-deterministic execution. If the server is deterministic, the problem is the same as for actively replicated servers. To see why, consider the following scenario: A primary replica sends a request to another server and then crashes. Once one of the backups has become the new primary it must resend the last request. To the invoked server, this is a duplicate request. Although this example illustrates a passively replicated server, a similar argument can be made for semi-passive replication.

Semi-active replication can treat the communication with another server in two different ways. First, the creation of the request could be treated as a non-deterministic operation. This would make the invocations identical, giving the illusion that the execution is deterministic, thus the duplicate invocation problem would arise. Second, the actual sending of the request could be treated as a non-deterministic operation and consequently only the leader would execute it. This would make the non-deterministic nature of semi-active replication opaque, as a failure of the leader could lead to a non-identical request being sent. This case is discussed in the following section.

5.1.2 The Orphan Invocation Problem

A non-deterministic passively replicated server will normally behave as shown in Figure 5.2. The primary of server C receives a request, processes it and invokes a second server D. When a reply is received, the primary updates the backups and returns a reply.

Unfortunately, a serious problem appears if the primary crashes [PKS03b, PKS03a]. To see why, consider Figure 5.3. If the primary of C sends a request to D and subsequently fails, one of the backups must become the primary. Because of the non-deterministic execution, there is no guarantee that the new primary will send an identical request to C. In fact, it may not invoke D at all, but rather a different server E, or it may not send the request (at all).

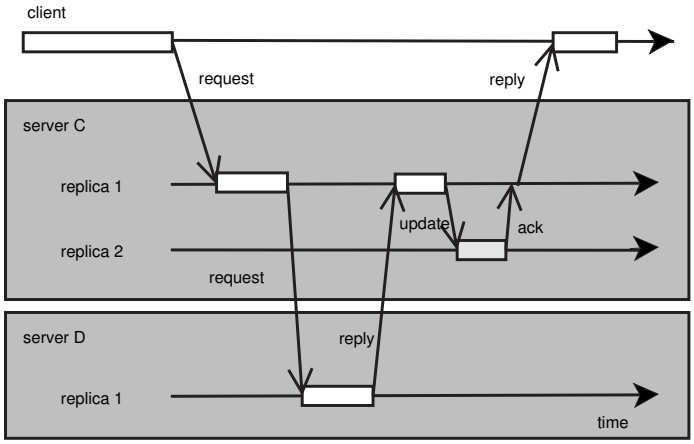


Figure 5.2: A non-deterministic passive server causing a replicated invocation

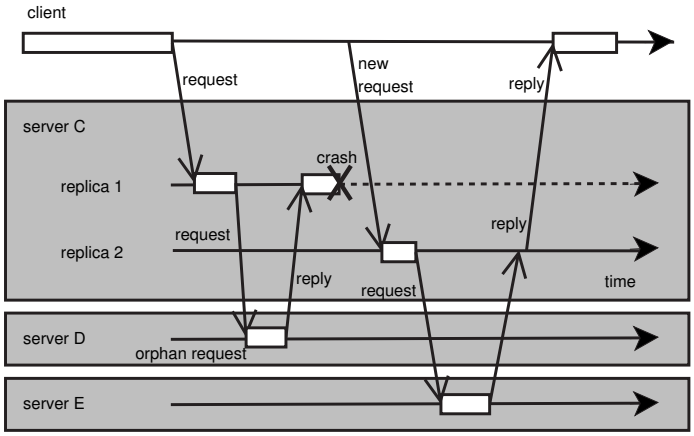


Figure 5.3: Non-deterministic passive replication and a primary failure lead to an orphan request

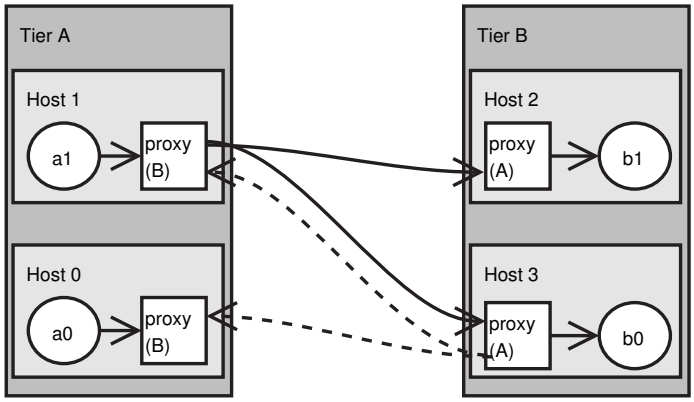


Figure 5.4: Pre-filtering and symmetric proxies [MGG95b]

If D processed the request received from the failed, old primary, it is currently in an inconsistent state. Even worse; a subsequent invocation of D can lead to a reply that leaks the inconsistency to the rest of the system. This behavior is not acceptable. The request leading to the inconsistencies is called an *orphan request*, and the problem that needs to be solved is the *orphan invocation problem*.

5.2 How to Handle Replicated Invocations

This section will give a short outline of how the difficulties associated with replicated invocations can be handled.

5.2.1 Duplicate Invocation Problem

The duplicate invocation problem is easily addressed: A timestamp for each outgoing message is generated. Since the servers are deterministic, all replicas produce identical timestamps and the server can easily filter duplicates by comparing them. The duplicate invocation by the two replicas of server A, shown as white arrows in Figure 5.1, will typically be filtered when they arrive at the (two) replicas of server B. Duplicate replies are also easily removed. Pre-filtering [MGG95a] at both the invoker and the invokee as shown in Figure 5.4 [MGG95b] can improve the scalability and performance. In the presence of failures, however, post-filtering at the invoker is also needed.

5.2.2 Orphan Invocation Problem

Handling the orphan invocation problem is less straightforward. The system must ensure that no orphans are allowed to cause inconsistencies. This can generally be done in three different ways in a replicated environment: Send undo information to backups, broadcast a new view, or use nested transactions. Chapter 6 suggests a fourth solution.

The (first) approach that sends undo information to the backups is explained in depth by Pleisch, Kupšys and Schiper [PKS03a]. It is based on a system

with no central transaction manager, but rather independent subtransactions that are terminated for each completed invocation. It differentiates between two types of subtransactions; optimistic and pessimistic. In the optimistic case, a subtransaction is committed as soon as it is finished, thus releasing all locks. It (optimistically) hopes that everything will work out. If something fails, the subtransaction must be compensated to restore the consistency of the system. In the pessimistic case, a subtransaction is not committed until its parent transaction commits. Thus, all locks are held indefinitely if the server executing the parent transaction fails. The solution for both cases is to send undo information to the backups before the invocation is sent. Thus, if the primary fails, the backup will be able to *terminate* unterminated pessimistic subtransactions and *compensate* terminated optimistic subtransactions.

The (second) approach that broadcasts a new view is a version of the orphan detection mechanism called *reincarnation* [TS01, pp. 380-381]. In the original approach, every client is given an *epoch* number that is sequentially increasing. Every time a client reboots, its epoch number is increased and broadcasted. When a server receives an epoch broadcast, all currently executing invocations from that client is aborted. Servers that do not receive the new view are notified when they reply to the client with an outdated client epoch number. For the replicated case, a new epoch begins when the leader of a group fails, and a new leader is appointed. This is the same as a view change [TS01, pp. 387-388] where the old leader is not in the new view. Thus, the new leader can broadcast the new view identifier, which can be used instead of an epoch number to kill off orphans. However, extending this to a partitonable environment is not straightforward, since partitioned concurrent views may exist.

The (last) approach [FN02] uses nested transactions and a central transaction manager. When a client sends an invocation to a server, a transaction is started. Each remote invocation from that server is treated as a subtransaction. If the parent transaction fails, the transaction manager rolls back all its orphan subtransactions. Consider Figure 5.5. Server C receives a request from the client, starts a transaction and sends an invocation to server D. A subtransaction is started, and upon completion it updates the backup. Then replica 1 of server C fails. Since the transaction manager is able to detect this as a part of its atomic commitment protocol, the rollback of the subtransaction at server D is initiated. The client then resends the request. This is done automatically, and is therefore transparent to the user. The figure also shows how the system is able to handle a failure at the leader replica of server D by failover to the backup.

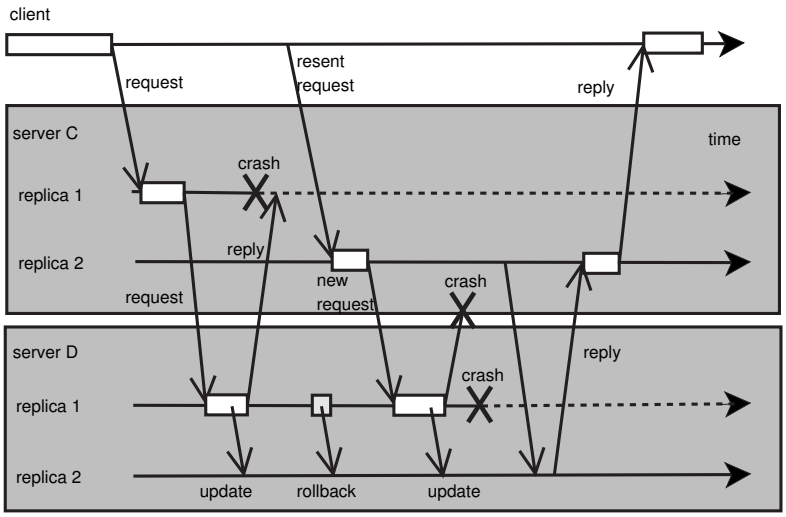


Figure 5.5: Two failures at two different and passively replicated servers using nested transactions

Chapter 6

Replicated Transactions

Transactions and replication are techniques to make a system fault tolerant, as outlined in Chapter 2. While the former yields consistency and safety, the latter provides availability and liveness. All of these are highly desirable properties. If the techniques are *combined* to work together instead of being provided as two separate services two additional properties can be achieved [FN02]:

- A "stronger consistency to replicated systems by supporting non-deterministic operation", and
- a "higher availability and failure transparency to transactional systems".

The former allows stronger consistency and non-deterministic execution because orphan requests can be rolled back in transactional systems. The latter provides higher availability and failure masking because replication allows the execution to be rolled forward to another replica. If exactly-once execution semantics is added on top of these, an available, consistent and failure-masking system that supports non-deterministic execution emerges.

The next sections show how the components involved in a transaction can be replicated. The replication of the transaction manager in a consistent way is explained in Section 6.1, and how to replicate the transaction participants and handling replicated invocations within a transaction are presented in Section 6.2

6.1 Replicating the Transaction Manager

Section 2.2 discussed why all single points of failure should 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 entire transactional system is rendered unavailable. This means that no transactions can be executed, and since the most widely used atomic commitment protocol, two-phase commit (2PC), may block if the TM becomes unavailable, it is imperative that the TM is replicated.

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 reality, TMs are not deterministic. In particular, timeouts introduces non-determinism since it cannot be guaranteed that all replicas timeout at exactly the same time [Pol93]. Also, active replication does not scale well, since executing the same processes on every replica wastes a lot of resources.

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 [GR93, BHG86]. In a distributed and non-replicated environment the decision is made persistent by force-writing a record to the log. In a replicated environment the transmission time is shorter than the time needed to write to the disk. A solution where the prepare decision is persisted by sending it to the backups is therefore both faster and preferred. In addition, it also gives better availability since the prepared transactions can be committed by the backup in case of a primary failure. If a local log was used, the transactions that are currently prepared may be blocked until the TM has recovered.

The decision to commit a transaction is also sent to the backups. This is done instead of the lazy write to the log in the normal non-replicated 2PC. Hence, the replicated nature of the TM is used to provide both availability and persistency of the decision.

6.2 Replicating the Transaction Participants

The transaction participants should also be replicated for the same reason as any other component of the distributed system; to avoid single points of failure. When replicated, they must be able to handle the problems occurring because of replicated invocations. As seen in Chapter 5, the actual problem and solution is dependent on whether the server is deterministic or not. When participating in a distributed transaction where passive replication is used, however, the problem is also dependent of whether the prepare request from the TM is allowed to failover or not.

A *failover* applies to passive replication and happens when the primary fails and an invocation is sent to the new primary instead. Figure 6.1 shows an example of a failover for a prepare request. First, the client tells the TM to commit a transaction. The TM then initiates the voting phase, and sends a prepare request to all primary servers that have joined the transaction. In this case, this is servers $A1$ and $B1$. The first has already failed, or fails while processing the request. The TM then resends the request to the new primary; $A2$. This is the failover. Then servers $A2$ and $B1$ replies with their votes, and the TM can continue the 2PC protocol, making its decision based on the received votes. The small arrow (3) shown in the figure is the mechanism that makes the commit decision persistent (durable) before replying to the TM.

Figure 6.2 illustrates what happens when there is no failover of the prepare request. When the TM does not get a reply from the failed leader, $A1$, it eventually times out and aborts the transaction (as indicated by Arrow 5).

With two independent binary variables there are four different designs that have to be analyzed:

- 1: Deterministic execution and failover for all requests.
- 2: Deterministic execution and no prepare request failover.

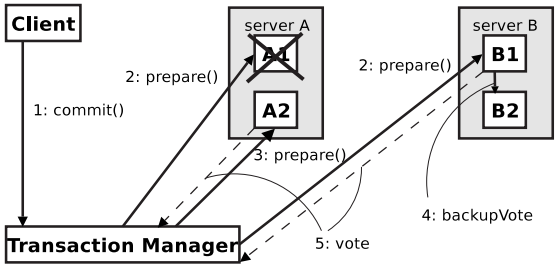


Figure 6.1: A failure of a primary, and the consequent failover

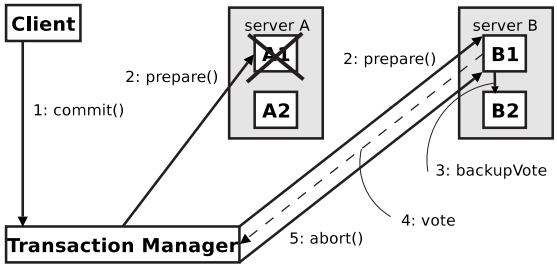


Figure 6.2: A failure of a primary, without the failover

- 3: Nondeterministic execution and failover for all requests.
- 4: Nondeterministic execution and no prepare request failover.

Depending on which of the cases above is chosen the problems to be solved differs: In the first scenario, the server replicas must be able to do duplicate detection and filtering as usual for deterministic systems.

The second design choice only makes sense if the group is passively replicated. However, it is not a particularly good one for that case either. Consider Figure 6.3: If replica $A1$ fails some time after sending message 7, but before the prepare phase of the transaction, the transaction will eventually timeout, be aborted and (probably) be restarted. This abortion might be unnecessary and it happens without any gain. If the prepare request was allowed to failover instead, the transaction would have been able to complete as long as a checkpoint was taken between the completion of the request from the client and the failure of $A1$. Then, $A2$ will have knowledge of the transaction and be able to complete the commit protocol. By checkpointing every time the server replies to a request, it can be *guaranteed* that the backup will be able to complete every transaction that the primary has joined.

As the two last choices both support nondeterministic execution they are in danger of creating orphan requests as seen in Section 5.1.2. For the third case, the problem will be the same as explained there, and the solutions are the same as presented in Section 5.2.2. This is illustrated by Figure 6.4: If replica $A1$ fails before updating the backup, $A2$, and replying to the client, the client will resend the request `doStuff1`. The execution of `doStuff2` will then be an orphan request at server B. This is due to the fact that the TM will get a reply from $A2$ in the prepare phase, and the failure of $A2$ will therefore go unnoticed.

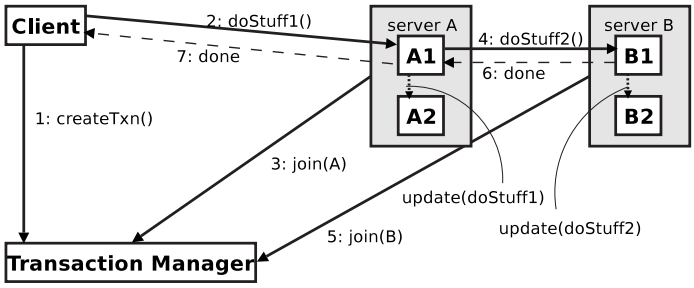


Figure 6.3: The creation and join phase of a transaction

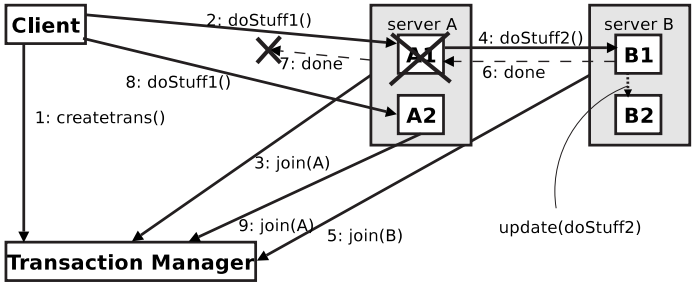


Figure 6.4: An unhandled orphan request

The transaction mechanisms described here cannot handle this situation alone.

The last case will be able to handle orphan requests. To see why, consider Figure 6.4 again. It shows the orphan request that is generated by the execution of `doStuff2` at server B and the subsequent failure of replica $A1$. When the client resends the original request to replica $A2$ in message 8, the system will have no knowledge of the orphan request, and it will continue towards transaction completion by joining A with $A2$ as the primary in the transaction, as though it was an original request in the failure free case. However, when the TM reaches its voting phase, replica $A1$, which is the primary of the first join, will not be able to vote. Thus, the voting phase will timeout, and the transaction, along with the orphan request, will be correctly aborted, as shown in Figure 6.2.

There is an obvious tradeoff between allowing the prepare request to failover or not. The first makes it harder to handle replicated invocations, while a failure of one participant in the latter causes the entire transaction to abort. Also, for the fourth and last case, the waste of resources might be substantial, because a transaction that is doomed to be aborted, is allowed to continue. However, by allowing the TM to break replication transparency and hence be able to remove the automatic failover for a replicated group, it provides a way for orphan requests to be handled correctly. There is no need for difficult and novel techniques such as those presented in Section 5.2.2.

6.3 Concurrency Control

Transactions that execute concurrently can interfere with one another and cause inconsistencies even though all of them executing serially is correct [BHG86]. Therefore, the result of a parallel execution must be equivalent to the result of a serial execution. The transaction history must be *serializable*. There exists many techniques for achieving this [BHG86]; two phase locking (2PL), timestamp ordering, serialization graph testing, certifiers and combinations of them. Since 2PL is the most common of these, it is presented in more detail below.

As the name indicates 2PL consists of two phases; one in which locks are set and one where locks are released. By never setting a lock after a lock has been released, the execution is guaranteed to be serial. There exist three different variations of 2PL: Basic, Conservative and Strict. The first suffers from possible deadlocks. If a transaction A holds a lock for object x and transaction B holds a lock for object y, and both transactions wants to set a lock for the other object, a deadlock has occurred.

Conservative 2PL avoids deadlocks by setting all locks at the start of the transaction. Thus, a transaction cannot request new locks after it has started processing, and deadlocks cannot occur.

The latter, Strict 2PL, is the variant most widely in use. It waits until the transaction commits to release any of its locks. Thus, a transaction will not read or write data written by an aborted transaction. This ensures that the histories are *strict*, which includes the highly desirable properties; *recoverable* and *avoid cascading aborts*.

In a distributed environment, global serialization is needed. However, by using Strict 2PL, each local site (or in this case; object group) knows that it can release the locks held by a transaction after it has received a commit message from the transaction manager. The rest of the thesis therefore assumes that serializability is ensured by Strict 2PL.

6.4 Ensuring Transaction Termination

When the TM is passively replicated as presented in this chapter, a transaction might be unable to terminate, and might block other transactions from completing. Consider the following case: A transaction has been created and all (or some) 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 it, 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 contact the participants telling them to abort the transaction. Because of possible replicated invocations each participant must be able to tell which other participants it has caused to joined the transaction,

and so on. However, if one of the participants also fails, the participants used 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.

Part III

Implementation and Assessment

Chapter 7

Implementation Issues

This chapter looks at the changes and additions to Mahalo needed to implement a passive replicated transaction manager. First, the package structure of Mahalo is presented, then the `PassiveGroupMahalo` (PGM), which is the transaction manager object being replicated, and the `GroupTxnManagerTransaction` (GTMT), which holds the state of the transaction, are explained in detail.

7.1 Overview

There already exists a replicated version of Mahalo—Gahalo [Mol04]. However, Gahalo is actively replicated, and as discussed in Section 6.1, a transaction manager should not be actively replicated, because of possible non-determinism. Also, active replication does not scale very well. Thus, a passive replicated TM is needed. The rest of this chapter gives a detailed explanation of the implementation of pGahalo - the passively replicated transaction manager. It is based on Moland's work [Mol04] on Gahalo, but extends it by allowing non-deterministic execution and controlling the replicated invocation related problems.

Before going into the technical details of the implementation in Section 7.3, an informal description of the life of a transaction in pGahalo is given in Section 7.2.

7.2 Informal Description

pGahalo is responsible for three different phases of the transaction: Its creation, the joining of transaction participants and its completion. These phases are presented in the following subsections.

7.2.1 Transaction Creation

Externally, transaction creation is almost done in the same way as under normal Mahalo (shown in Figure 4.2). The only external difference is that the `TransactionFactory` invokes pGahalo using *leadercast semantics* (see Section 4.2.5).

When pGahalo receives a request to create a new transaction, as shown in Figure 7.1, it forwards it to the `GroupTxnManagerImpl` (GTMI). It is responsible

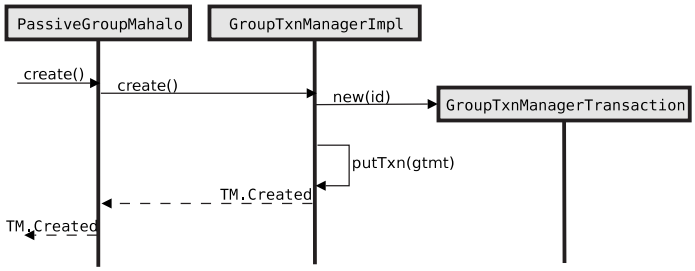


Figure 7.1: The creation of a new transaction in pGahalo

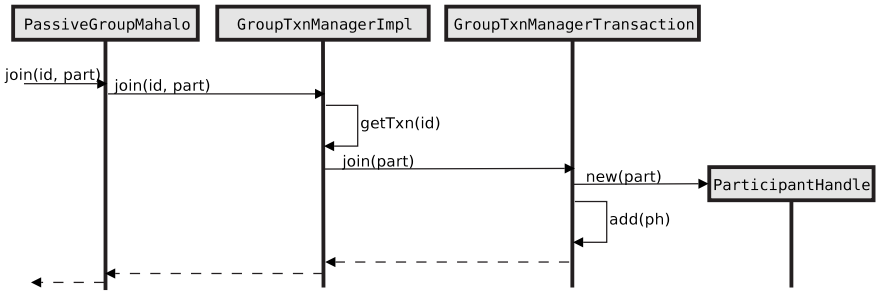


Figure 7.2: The join of a participant in a transaction in pGahalo

for creating a new transaction identificator and a new GTMT object, which is the object containing the state of the transaction. The new object is then stored in a hashmap, where the transaction identificator is used as a key. Then an object with the identificator and the information about the lease is returned to the factory and client through PGM.

7.2.2 Joining the Participants

As for transaction creation, the leadercast invocation on the TM is the only external difference between pGahalo and Mahalo when joining the participants (see Figure 4.2). The more detailed Figure 7.2 shows how pGahalo joins a participant in a transaction. PGM receives a join-request from a participant, which contains the identificator of the transaction and a proxy for the participant. It is passed on to the GTMI, which retrieves the correct GTMT from the hashmap using the identificator. The participant proxy is sent to the found GTMT, which makes a new `ParticipantHandle` and adds it to its list of handles. The execution then returns to the invoking server.

In this prototype implementation of the transaction participants no concurrency control, as explained in Section 6.3, is implemented. The banks simply execute their operations in the order they are received.

7.2.3 Transaction Completion

Both Mahalo and pGahalo uses the two phase commit presumed abort protocol (2PC-PA) [MLO86] to atomically terminate transactions. The reasons for choosing this protocol over the two other variants of 2PC, presumed commit and

presumed nothing, are twofold. First, it is very easily fitted to the replicated nature of pGahalo. If the primary fails and the backup knows nothing about the transaction, the participant can safely abort the transaction. Second, because of how Mahalo is implemented, it is easier and faster to implement 2PC-PA than the others.

There are limitations for this implementation of 2PC-PA. These are caused by shortage of time and a main goal to construct a prototype system to demonstrate, test and give an indication of the performance. The implementation assumes that there will always be at least one working replica of the transaction manager. The outcome of each transaction is made persistent by replication, not by logging. However, if all replicas fail, there is no log from which to do recovery. Similarly, if all replicas of a transaction participant fail, its state will be lost. However, all running transactions will still be able to terminate.

No optimization of the commit protocol has been implemented. For instance, it is not possible for a participant to give a read vote, which would exclude it from the second phase of 2PC, and 2PC is performed separate from the operations of the transactions. In particular, there is no piggybacking of prepare and commit messages on the transaction invocations. In addition, all messages are synchronous. Therefore, the client will not receive any early commit messages.

Figure 4.3 shows how a successful commit of a transaction works. For pGahalo it starts by receiving a commit request, which is forwarded to the GTMI. It retrieves the correct GTMT from the hashmap based on the identificator that came with the request. For each participant in the list of `ParticipantHandle`s, a `PrepareJob` is scheduled and executed. If all returned yes-votes, GTMT multicasts a message to all PGMs, hence persistently storing the decision to commit. The message is sent using an internal group method invocation (IGMI, explained in Section 4.2.5. The backups of the PGM are shown as grey lines the figure.

After receiving confirmation from the backups saying that they are aware of the decision to commit, GTMT initiates a `CommitJob` for each participant. When they are finished, GTMT sends the information to all PGM, telling them that the transaction has completed. Thus, the transaction (the GTMT object) can be removed from the map of transactions, and the client can be notified of its successful completion.

If the result of one or more `PrepareJob`s is a no-vote, then the GTMT invokes the `abort()` method of GTMI (see Figure 7.4. After the abort has finished as explained next, a `CannotCommitException` is thrown back to the client, which would have to handle it in a suitable way.

A transaction abort can be initiated by one of the participants voting no in the prepare phase as seen in Figure 7.4, or if, for some reason, the client decides to abort it. The PGM then receives an abort request that is forwarded to the GTMI object as shown in Figure 7.5. It retrieves the correct GTMT object from the transaction hashmap and calls its `abort()` method. Then, for each participant that has joined the transaction, an `AbortJob` is started. After all jobs have completed the GTMI removes the transaction from the hashmap and returns. There is no need to notify the backups, since the 2PC-PA protocol is used.

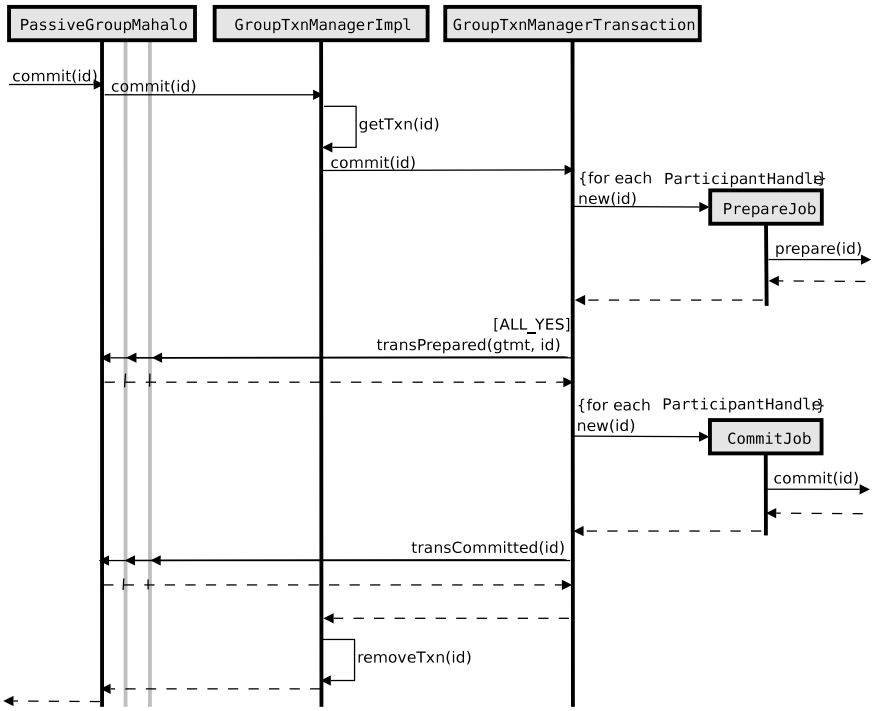


Figure 7.3: A transaction commit in pGahalo

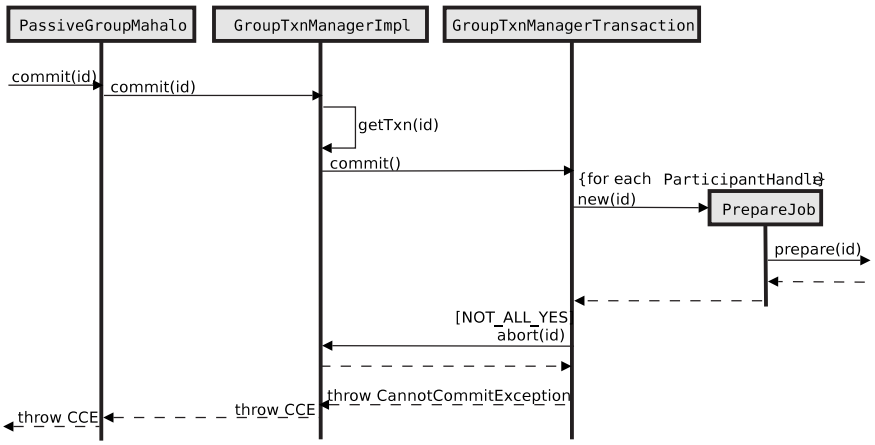


Figure 7.4: A participant caused transaction abort in pGahalo

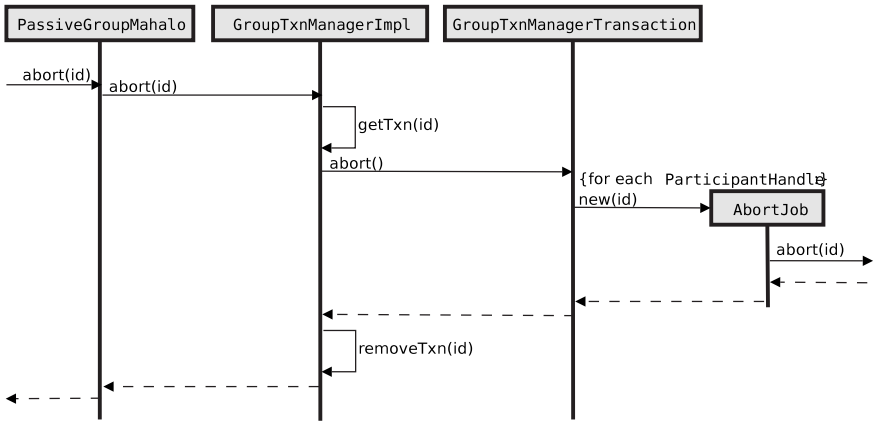


Figure 7.5: A client initiated transaction abort in pGahalo

7.3 Technical Details

The next subsections first present the package structure of Jini. Second, the changes needed to deny failover for the prepare request are explained. Then, the details on how pGahalo and the passively replicated transaction managers were implemented are given, and finally, the implementation of a bank that can participate in transactions is outlined.

7.3.1 Package Structure

All packages in Jini 2.0 are either on the form `net.a.b.c` or `com.a.b.c`. All the net-packages are public, while the com-packages are private.

The main package structure of Mahalo is as follows:

- `com.sun.jini.mahalo`: The core implementation of Mahalo
- `com.sun.jini.mahalo.log`: The implementation of the logger used in Mahalo
- `net.jini.core.transaction`: Support classes (transaction factory and exceptions) for Mahalo
- `net.jini.core.transaction.server`: The external and public interfaces implemented by the classes in `com.sun.jini.mahalo`.

7.3.2 Adding support for bypassing failover

To be able to support nondeterministic execution for the transaction participants, the failover of the prepare request must be denied as explained in 6.2. The leadercast semantics in Jgroup/ARM is designed to failover to the new primary. This failover mechanism needs to be bypassed when needed.

First, the changes visible to the application programmer are explained, then the changes to the internals of Jgroup/ARM needed to accommodate the semantics of the addition to the `Leadercast` interface are presented. Two things

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Leadercast {
  boolean failover() default true;
}
```

Figure 7.6: The `Leadercast` annotation interface

are needed to make it a general feature that is available for an application programmer using Jgroup/ARM:

- A way to mark a method that should not failover.
- The actual implementation that makes the bypass possible.

To provide the application programmer with a way to mark the methods that should not failover, the `Leadercast` annotation type was changed. The boolean return type `failover` was added, seen at Line 5 in Figure 7.6. A method that needs to make sure that only the primary receives an invocation, despite of failures, only needs to prefix the method with `@Leadercast(failover = false)`. The default value is `true`, hence if only `@Leadercast` is specified normal failover will occur.

In Jgroup/ARM the `GroupInvocationHandler` (GIH) is responsible for performing the external group method invocations on the client side. A `GroupEndPoint` (GEP) holds the references to all endpoints of a group proxy, and it is also responsible for updating the list of endpoints by contacting the registry if an invocation should fail. When a new external group method invocation (EGMI) is executed, the GIH contacts the GEP, which is responsible for selecting the endpoint depending on the invocation semantics of the invoked remote method. For instance, if the method is invoked with leadercast semantics, only the leader is chosen, or if anycast semantics is used, a randomly chosen endpoint is picked. The endpoint is returned to the GIH, which tries to execute the invocation. If it fails (by throwing an `IOException`), the endpoint is removed from the list of endpoints, and the GEP selects a new endpoint. This is the failover mechanism.

The failover mechanism can be bypassed by a simple test: When an `IOException` is caught and leadercast semantics is used, check the failover variable for the semantics. If the variable is `false`, throw a `GroupUnreachableException`, else continue as normal. Thus, the failover for leadercast semantics can be disallowed for the prepare request.

7.3.3 Implementing pGahalo

pGahalo is the implementation of a passively replicated transaction manager. It is based on the Jini transaction manager, but augmented to be able to conform to the transaction manager described in Section 6.1.

The changes needed for the main transaction manager class and the class controlling the state of each transaction is presented and explained in the following subsections.

```
public interface TransactionManager extends Remote, TransactionConstants {

    Created create(long lease) throws LeaseDeniedException, RemoteException;

    void join(long id, TransactionParticipant part, long crashCount)
        throws UnknownTransactionException, CannotJoinException,
            CrashCountException, RemoteException;

    int getState(long id) throws UnknownTransactionException, RemoteException;
                                                                          10
    void commit(long id)
        throws UnknownTransactionException, CannotCommitException,
            RemoteException;

    void commit(long id, long waitFor)
        throws UnknownTransactionException, CannotCommitException,
            TimeoutExpiredException, RemoteException;

    void abort(long id)
        throws UnknownTransactionException, CannotAbortException,          20
            RemoteException;

    void abort(long id, long waitFor)
        throws UnknownTransactionException, CannotAbortException,
            TimeoutExpiredException, RemoteException;
}
```

Figure 7.7: The `TransactionManager` interface

The PassiveGroupMahalo class

The main class of pGahalo is `PassiveGroupMahalo` (PGM). It is the class that implements both the EGMI and IGMI interfaces. The first allows the public methods in PGM to be accessed via remote methods from other servers. These methods are the same that are declared in the Jini `TransactionManager` interface (see Figure 7.7). The latter is done indirectly by specifying and implementing the `InternalPassiveGroupTransactionManager` (see Figure 7.8), and it provides the necessary means to internally (to the group) notify the backups of the prepare and commit of each transaction.

The invocation semantics of each method can (as explained in 4.2.5) easily be set by prefixing the method declaration with metatags. To ensure that only the primary gets the invocations from the clients and participants, all implemented methods in PGM from the `TransactionManager` interface is prefixed with "`@Leadercast`", to achieve leadercast semantics. The prepare method is prefixed with `@Leadercast(failover = false)` as explained in the previous section.

```
public interface InternalPassiveGroupTransactionManager
  extends InternalGMIListener, java.io.Serializable
{
  public void transPrepared(TxnManagerTransaction tmt, long id)
    throws RemoteException;

  public void transCommitted(long id)
    throws UnknownTransactionException, RemoteException;
}
```

Figure 7.8: The `InternalPassiveGroupTransactionManager` interface

The GroupTxnManagerTransaction class

The `TxnManagerTransacion` (TMT) class is responsible for the state of a transaction in Mahalo. To be able to support replicated transactions, however, some changes were needed.

First of all, the TMT class in Jini 2.0 implements the `Serializable` interface, but unfortunately some of its instance variables are not serializable[1]. Thus, the TMT could not be sent over the network.

Second, to be able to backup the transaction at the end of the prepare phase and the commit phase, two new methods were needed; `allParticipantsPrepared()` and `allParticipantsCommitted`, respectively. The first is invoked from the `commit()`-method after all participants has prepared, while the latter is invoked at the successful completion of the same method. Both of the new methods retrieve a reference to the IGMI service for pGahalo (see Figure 7.8) through a static method call to the `PassiveGroupMahalo` class. Then, using the methods defined as the IGMI service, they notify the backups of the new state of the transaction.

Because of the two issues above, a new class that inherits TMT is needed; `GroupTxnManagerTransaction` (GTMT). It overwrites the `commit()`-method and adds the two new ones. Also, the non-serializable variables are marked as *transient*, meaning that they do not get serialized (marshaled) when the GTMT object is sent over the network. Thus, these variables have to be reinitialized when received by the pGahalo backups. Because the TMT (and the GTMT) is part of the private com-classes, this is done using *reflection* [Bar01].

7.3.4 Implementing the Replicated Transaction Participants

By disallowing failover for the prepare request the orphan invocation problem for nondeterministic servers (presented in Section 5.2.2) can be avoided. The details were explained in Section 7.3.2. Thus, we can allow nondeterministic servers as transaction participants. The only change that is needed for the application is to add a value to the `Leadercast` annotation interface.

An implementation of a transaction participant, the `ReplicatedBankServer` is presented here. It is passively replicated and supports nondeterministic exe-

[1] This is reported as a bug at the Sun Bug Database and is now fixed for the next release http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4912745.

```
public interface TransactionParticipant extends Remote, TransactionConstants {

    int prepare(TransactionManager mgr, long id)
      throws UnknownTransactionException, RemoteException;

    void commit(TransactionManager mgr, long id)
        throws UnknownTransactionException, RemoteException;

    void abort(TransactionManager mgr, long id)
        throws UnknownTransactionException, RemoteException;

    int prepareAndCommit(TransactionManager mgr, long id)
        throws UnknownTransactionException, RemoteException;
}
```

Figure 7.9: The `TransactionParticipant` interface

```
public interface InternalPassiveTransactionParticipant
  extends InternalGMIListener, java.io.Serializable
{
  public void txnPrepared(long id, Object stateUpdate)
    throws RemoteException;

  public void txnCompleted(long id, int outcome)
    throws UnknownTransactionException, RemoteException;
}
```

Figure 7.10: The `InternalPassiveTransactionParticipant` interface

cution. The extensions made to the standard Jini transaction participants are explained, and a

The ReplicatedBankServer class

Any implementation of a transaction participant in Jini must implement the `TransactionParticipant` interface shown in Figure 7.9, since the transaction manager must be able to run the 2PC protocol. To be able to replicate it and use EGMI the participant must implement the EGMI interface. The methods available for IGMI are declared in an interface which it must also implement.

Two IGMI methods are needed for a participant to notify the backups of two events. A primary yes vote as a response to a prepare request from the transaction manager and a transaction commit or abort decision. The `InternalPassiveTransactionParticipant` interface is shown in Figure 7.10.

The EGMI methods are the methods declared in the `TransactionParticipant` interface, plus any application specific methods. For instance, a bank would probably have methods for withdrawals, deposits and reading the balance of an account. A participant would join the transaction when one of its methods was called with a transaction identificator. The join method of the transaction

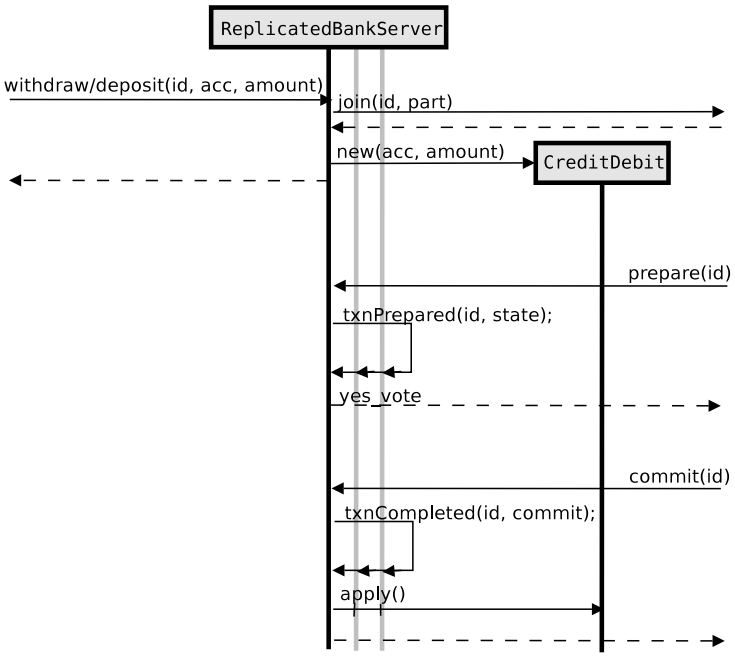


Figure 7.11: The commit of the `ReplicatedBankServer`, an implementation of a `TransactionParticipant`

manager would be called with two parameters, the transaction identificator and a proxy. The latter is needed by the manager to be able to contact the participant as a part of 2PC.

The implementations of the EGMI methods are prefixed with `@Leadercast`. To facilitate nondeterministic execution the `failover` variable is set to "false" for the prepare method, as explained in Section 7.3.2. The rest of the methods use, and need, the failover mechanism.

Figure 7.11 illustrates the successful completion of a transaction as seen from a transaction participant. The implementation used in this thesis is called the `ReplicatedBankServer`. It provides methods to withdraw and deposit a certain amount of money from and into an account as a part of a transaction. When it is invoked with an operation it joins the transaction by sending the transaction identificator and a group proxy to the TM. Then a new `CreditDebit` object containing the operation is created before a reply is sent to the client.

After a while, the other operations of the transaction has been executed and the TM has started its voting phase, the participant will receive a prepare message from the TM. Then, if it is ready to commit, the transaction and its operations is made persistent by using the IGMI `txnPrepare()`. A yes vote is returned to the TM if the IGMI completed successfully.

Later, the participant will receive the decision from the TM. The decision is sent to all backups, and if the decision is to commit, as in the figure, all of the replicas apply the `CreditDebit` object to their state. A reply is sent from the primary to the TM containing an acknowledgment of the transaction outcome.

Chapter 8

Tests

This chapter presents the environment used for testing and the results of tests executed on the pGahalo transaction manager. The tests include comparisons of response times between pGahalo, Gahalo, and Mahalo as well as failover measurements for pGahalo.

8.1 The Test Environment

The system where the tests are executed consists of four conceptual entities: A client, a transaction manager (TM) and two banks. Figure 8.1 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 lifecycle of the transaction used for testing is as follows:

- A transaction is initiated by the client, and created by the TM.
- The client invokes the withdraw operation of Bank_A, which joins the transaction.
- The client invokes the deposit operation of Bank_B, which joins the transaction.
- The client initiates 2PC, which is carried out by the TM as explained in Section 7.2.3.

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 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

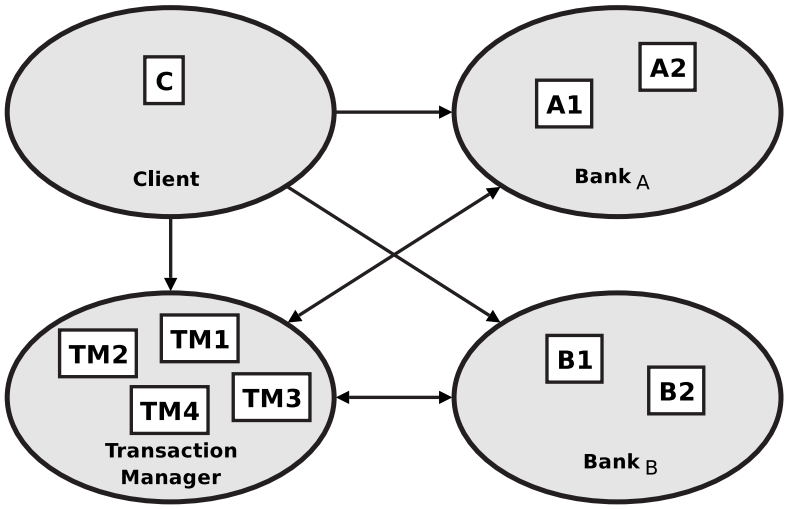


Figure 8.1: A model of the system used for testing

response time of an invocation is the time passed between calling the remote method of the client and the return of the method call.

All results show that there is a startup cost at the beginning of each testrun that affects the first transactions executed. After about 50 transactions the response time normalizes. The startup cost is probably due to the allocation of objects and the dynamic run-time compilation of Java. The latter optimizes code that is very frequently accessed such that it executes faster. The graphs presenting the response times clearly show this. The first 50 transactions are therefore left out of all histograms and any calculations made, and ignored in the discussion.

Failover measurements are executed with the configuration of variables as shown in Table 8.1.

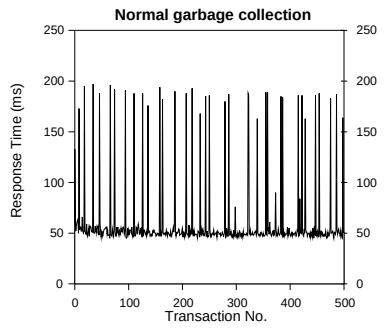
8.2 The Issue of Garbage Collection

All testruns presented in this chapter are done using incremental garbage collection (GC). This is specified by using `-Xincgc` as an argument to the Java Virtual Machine when starting the application. This causes the GC to run continuously to remove dereferenced objects and class definitions. Normally, GC is performed at more or less regular intervals when the memory is full. It preempts the execution of the normal services and leads to increased response times for the affected transactions.

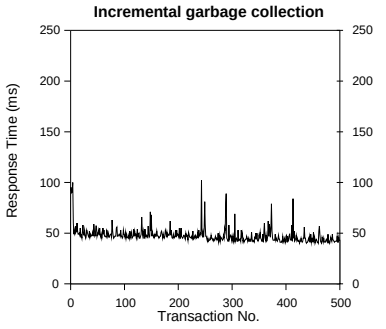
Figure 8.2 shows the effect of using incremental GC instead of the default. It illustrates that a normal GC blocks the transaction processing for about 150 ms. This causes the response time to be four times longer for the nonreplicated case. As will be presented in the following sections, a transaction executing in a replicated system will take longer to execute. The relative effect of a single GC is therefore less. However, the probability for a transaction to be delayed at more than one point is increased, which might cause an even higher response time for a transaction.

Variable	Description	Value
`routing-Timeout`	This variable is the time that a node waits for a heartbeat message from another node before it times out. If it is set too low, too many false suspicions of a failure happens. On the other hand, if it is set to high, failover will take a long time.	50
`maxTTL`	This variable decides how many times a server can time out before it is suspected to have crashed. The consequences are the same as for the previous variable	5
`alfa`	This variable automatically tunes the timeout value. This is in principle the same as TCP's exponential weighted moving average calculation. However, tuning this variable did not yield any significant change for the failover delay.	7/8

Table 8.1: The configuration of the variables tuned for failover measurements



(a) Response times for normal garbage collection



(b) Response times for incremental garbage collection

Figure 8.2: Transaction response time graphs comparing two kinds of garbage collection. Both are performed without replication.

Garbage Collected	Objects	Size
Nonreplicated system		
Primary	2.2M	76MB
2 passive TM replicas		
Primary	4.4M	123MB
Backup	3.9M	106MB
3 passive TM replicas		
Primary	5.3M	146MB
Backups	4.4M	117MB
4 passive TM replicas		
Primary	6.2M	168MB
Backups	4.8M	130MB

Table 8.2: The effect of replication on garbage collection

Depending on the transaction class this uncertainty may be too high. For instance, a user executing requests over the Internet will probably never notice it. In a telecom-system, however, such delays can be unacceptable.

Figure 8.2(b) shows that there are some peaks when using incremental GC as well, but they are fewer and smaller. The origins of these peaks are hard to identify, and can also be caused by operative system processes running or network delays.

The number of garbage collected objects while executing transactions can be found by using the YourKit Java Profiler[1]. 500 transactions were executed in four separate testruns for three degrees of replication. The results are shown in Table 8.2. It clearly shows that there is a large increase in the number of garbage collected objects and the total size of the objects when the transaction manager is replicated.

8.3 Costs of Passive Replication

Ideally, the cost of executing a replicated service should be zero. Since replication has to be managed and controlled, and these threads have to compete with the execution of transactions for resources, however, a performance penalty is unavoidable. In addition, multicast is generally slower than unicast [MMBH02], causing an extra communication delay.

Section 8.3.1 presents the results of measurements done on both partially and fully passively replicated systems and compares the resulting response times observed from the client with those of a nonreplicated one. Similarly, Section 8.3.2 gives a comparison between active and passive replication.

8.3.1 Passive Replication versus No Replication

By comparing the measurements for the nonreplicated case, with those of varying degrees of passive replication, an understanding of how it affects the per-

[1] www.yourkit.com

formance can be gained. Therefore, measurements were done not only for the fully replicated case, but also for partially replicated cases.

First, only replication of the transaction manager is considered and then replication of the transaction participants is added.

Replicating the Transaction Manager

The passively replicated pGahalo was replicated with degrees varying from two to four, and client-side response times were collected for each transaction. The following runs were performed:

- Testrun 1: A nonreplicated transaction manager and nonreplicated banks.
- Testrun 2: Two passively replicated transaction managers and nonreplicated banks.
- Testrun 3: Three passively replicated transaction managers and nonreplicated banks.
- Testrun 4: Four passively replicated transaction managers and nonreplicated banks.

The purpose of these testruns was to see how much overhead is added by passive replication of the transaction manager, and how much the variance of the response times increases. Figure 8.3 presents the results of these runs.

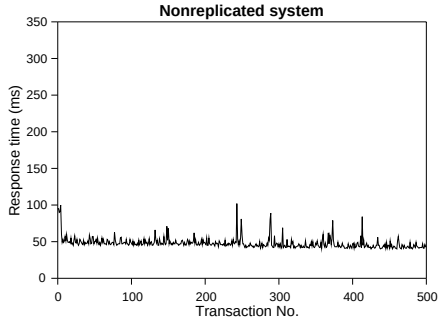
The response time of Testrun 1 in Figure 8.3(a) is pretty stable. There are about eight relatively small peaks spread randomly out. These peaks are probably caused by the incremental garbage collection being extra active at the time, or a delay in the network traffic. Also, operative system processes might have interfered in the execution.

Figure 8.3(b) illustrates the execution time for Testrun 2. It shows more unreliable response times than the nonreplicated run. When utilizing the Jgroup system, there are a lot more objects that need to be garbage collected as explained in Section 8.2. Also, there is more network traffic, which can incur delays, and the transactions have to compete with the threads managing the replication. All of these increase the uncertainty for the response time of a transaction. The average response time is also longer because of the overhead added by Jgroup and the multicast of the prepare and commit decisions.

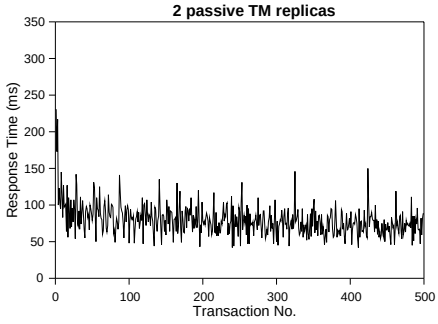
Testrun 3 and 4 shows even higher response time variance. These are caused by the same reasons as those mentioned for Testrun 1, but their impacts are larger, because of a higher degree of replication.

Figure 8.5 shows the distribution of the response times for each of the testruns. The histograms seem to be skewed towards shorter response times when increasing the degree of replication. There is a much clearer lower limit for the nonreplicated case, than for the replicated ones. This is caused by an increased probability of a transaction delay in a more complex system with more nodes. With a lower probability of delays, more transactions will be completed closer to the lower limit. Only a small increase in variance is observed as more replicas are added.

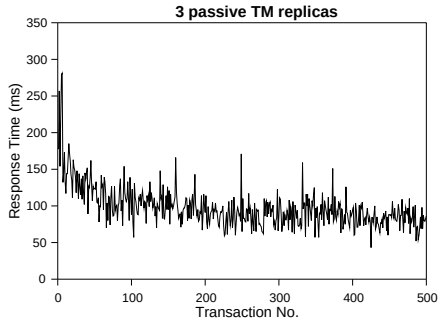
As can be estimated from Figure 8.4, the average response time is a bit over 50 percent greater when using two passive replicas of the TM compared to the



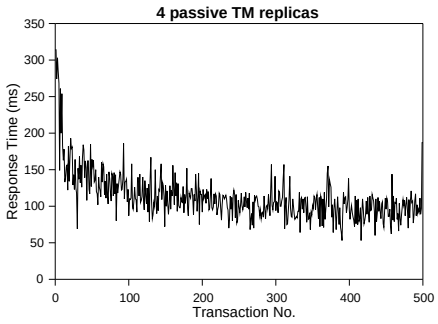
(a) Nonreplicated system



(b) 2 passive TMs



(c) 3 passive TMs



(d) 4 passive TMs

Figure 8.3: The cost of passively replicating the transaction manager

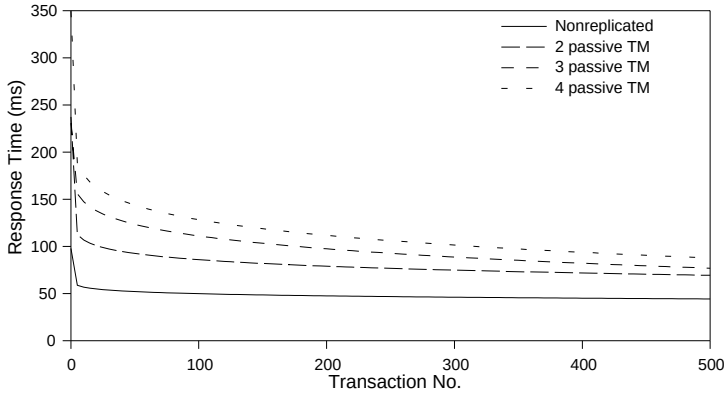
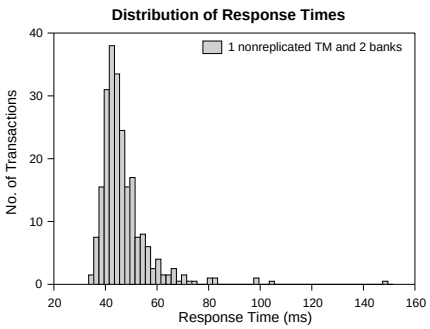
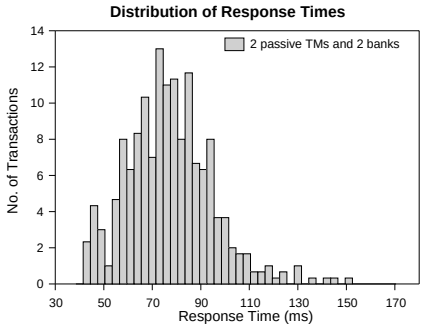


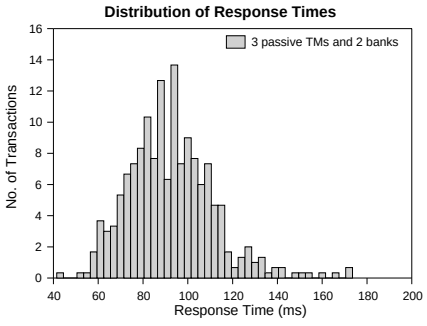
Figure 8.4: Regressions of the plots in Figure 8.3



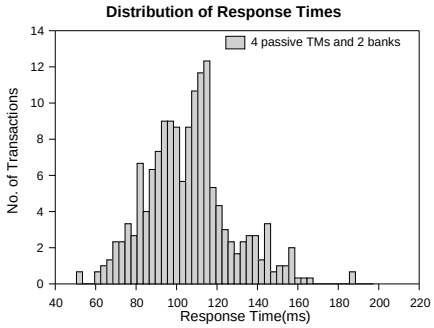
(a) Nonreplicated system (2 ms)



(b) 2 passive TMs (3 ms)



(c) 3 passive TMs (3 ms)



(d) 4 passive TMs (3 ms)

Figure 8.5: Histograms of the distribution of response times for various degrees of transaction manager replication. The number in parentheses represents the horizontal size of the bars.

nonreplicated case. Three and four replicas cause around a 100 and 150 percent increase, respectively.

Fully Replicated System

The transaction participants were replicated along with the transaction manager in these testruns to create a fully replicated system. The purpose was to find the overhead of replicating the transaction participant and creating a fully replicated system, and also to see if this adds any variance to the results. The following testruns were performed to be able to examine this:

- Testrun 5: A nonreplicated transaction manager, with two replicas of each bank.
- Testrun 6: Two passively replicated transaction managers, with two replicas of each bank.
- Testrun 7: Three passively replicated transaction managers, with two replicas of each bank.

Figure 8.6 shows the response times for these testruns. Testrun 1 is plotted as well in Figure 8.6(d) for easier comparison.

Replicating the transaction participants results in more peaks and generally a shorter response time. The peaks are probably caused by the same reasons as the added variance for the replication of the transaction manager; added network traffic and replication management. The added delay is because of multicast and system complexity.

For Testruns 6 and 7, the overhead and variance increase even more. This is of course caused by the increased complexity of the system, plus some extra might be added because of some nodes executing more than one service because of only the limited number of available nodes for testing. Also, updating the backups is considerably slower as discussed in Section 9.1.1.

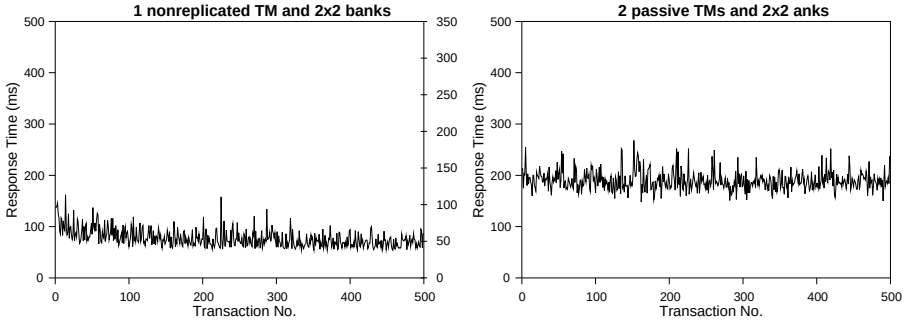
Looking at Figure 8.7 we observe the same phenomena as in Figure 8.5: Increasing the degree of replication causes more variance and the top of the graphs moves away from the lower bound.

Figure 8.6(d) shows a regression of the response time for Testruns 1 and 5-7. It indicates that replicating the banks causes a 60 percent increase in the response time. Adding a replicated transaction manager on top of that causes a 100 percent increase. Using three replicated transaction managers and two replicated banks is more than two and a half times as time consuming as only replicating the banks, and around four times slower than the nonreplicated case.

8.3.2 Passive Replication versus Active Replication

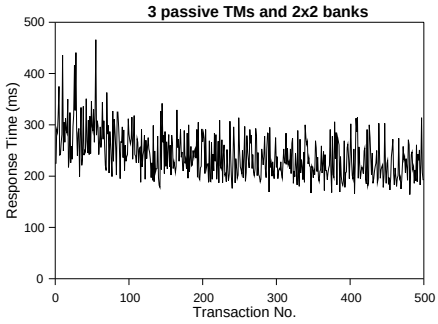
To evaluate pGahalo properly, its overhead needs to be compared to that of Gahalo. Therefore, the following testruns were performed using Gahalo:

- Testrun 8: Two actively replicated transaction managers and nonreplicated banks
- Testrun 9: Three actively replicated transaction managers and nonreplicated banks

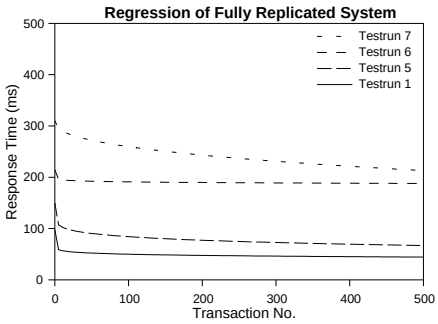


(a) 2 replicas of each bank and 1 passive TM

(b) 2 replicas of each bank and 2 passive TMs

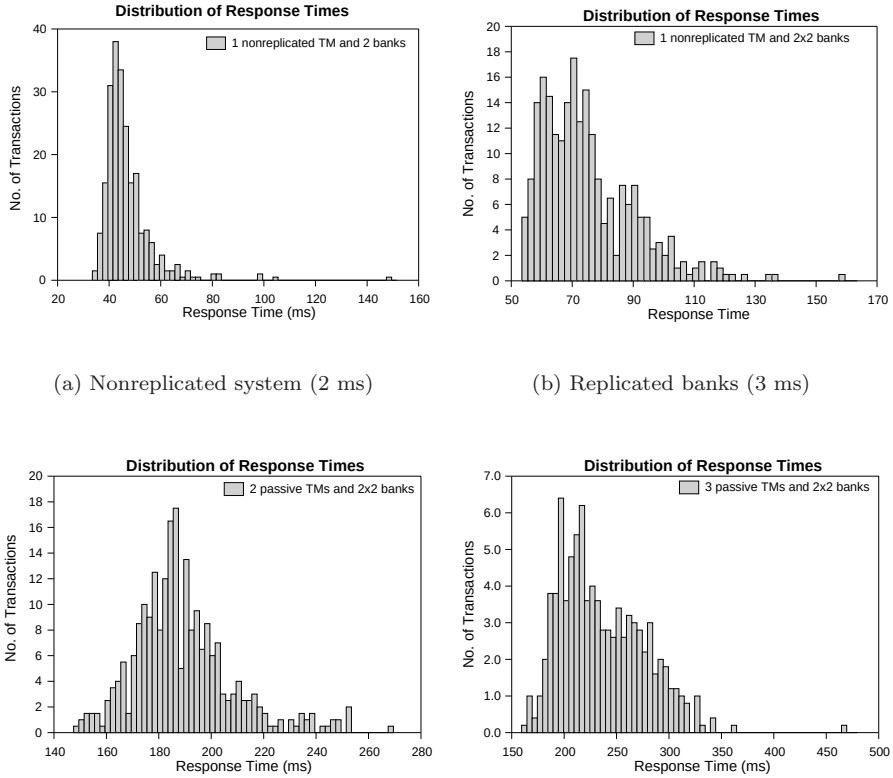


(c) 2 replicas of each bank and 3 passive TMs



(d) Regression of Testruns 1 and 5–7

Figure 8.6: The cost of actively replicating the transaction manager



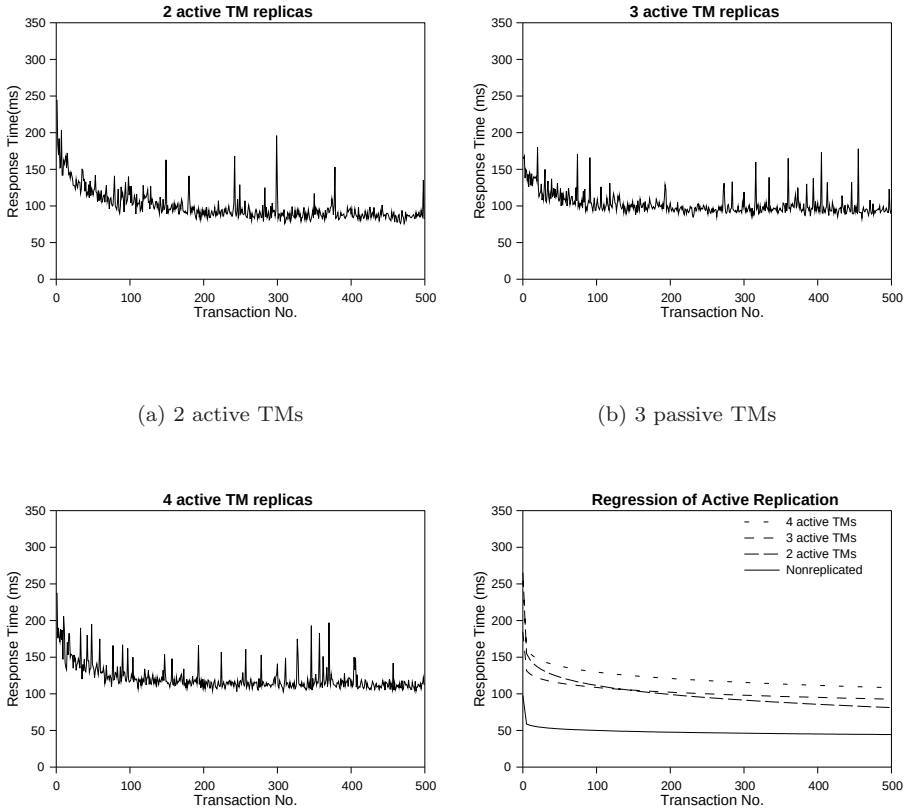
(a) Nonreplicated system (2 ms)

(b) Replicated banks (3 ms)

(c) Fully replicated with 2 passive TMs (3 ms)

(d) Fully replicated with 3 passive TMs (6 ms)

Figure 8.7: Histograms of the distribution of response times for various degrees of replication. The number in parentheses represents the horizontal size of the bars.



(a) 2 active TMs

(b) 3 passive TMs

(c) 5 active TMs

(d) Regression of Testruns 1 and 8–10

Figure 8.8: The cost of actively replicating the transaction manager

- Testrun 10: Four actively replicated transaction managers and nonreplicated banks

These testruns are plotted in Figure 8.8. The plot from Testrun 1 is added in Figure 8.8(d) for easier comparison with the passive test results shown in Figures 8.3 and 8.4.

There are some points worth noticing when comparing the figures.

- Active replication seems to normally yield less variance.
- Active replication seems to produce higher max values in the response times.
- The response times for passive replication are, in average, shorter than for active replication.

The first point is because of Jgroup's handling of active replication. The client receives replies from all servers, but only the first reply is required to

continue the execution. Thus, if all but one server is delayed, the client still receives a fast answer.

The second point is explained by careful examination of the log. Each of the highest response times for Gahalo, all between transaction numbers 300 and 400, is caused by two separate delays in two separate remote operations. Thus, this effect is not a general property of Gahalo, but rather of the specific testrun.

The third and last point is probably caused by the fact that multicast is generally slower than unicast, and Gahalo has more multicasts per transaction than pGahalo.

8.4 Failover Delay

The downsides of using passive replication for the transaction manager are twofold. They are both related to primary failures. If the primary fails, two things happen:

- Transactions that have not prepared are aborted.
- A *failover* delay until the backup takes over the processing, new transactions cannot be created and prepared transactions cannot be terminated during this timeframe.

The first depends on the number of clients and transactions per client in the system at the time of failure. Since only one client is running serial transactions in the testruns, this will be maximum one transaction. The real failover delay is not possible to measure since one cannot accurately tell the time that another node in the system became unavailable. This is because one cannot reliably discern a slow process from a crashed process in an asynchronous system [FLP85]. Thus, there is no way to measure the exact time to elect and initiate a new primary.

The downtime as observed from the client, however, can be measured. It is done by logging the response time of each remote operation. When a primary fails the increased response time can be read from the log. Since a primary failure is a rare event, however, the failures were inserted by manually sending a stop signal to the process.

Figure 8.9 shows a testrun where 3 passive transaction managers and two nonreplicated banks were started. After nearly 200 transactions the primary was killed, and after about 370 transactions the new primary was also killed.

A careful study of the figure reveals some interesting aspects. After the initial 50 transactions the response time normalizes until the first failure. However, just after the first failure there are a few slower transactions while the system adjusts to the new environment. The response time then stabilizes again. The second failure causes a few slower transactions, then the response time is relative stable for the rest of the testrun. The two small peaks after about 150 and 420 transactions can be caused by any of the reasons given in Section 8.3.1.

The peaks for both failures are just below 500 ms. The failover delays, however, are about 400 ms since the transactions normally takes around 100 ms to complete in a failure-free case with replicated TMs. This result was verified by executing 50 failovers. The results are shown in Figure 8.10. They showed the same characteristics as the example presented here.

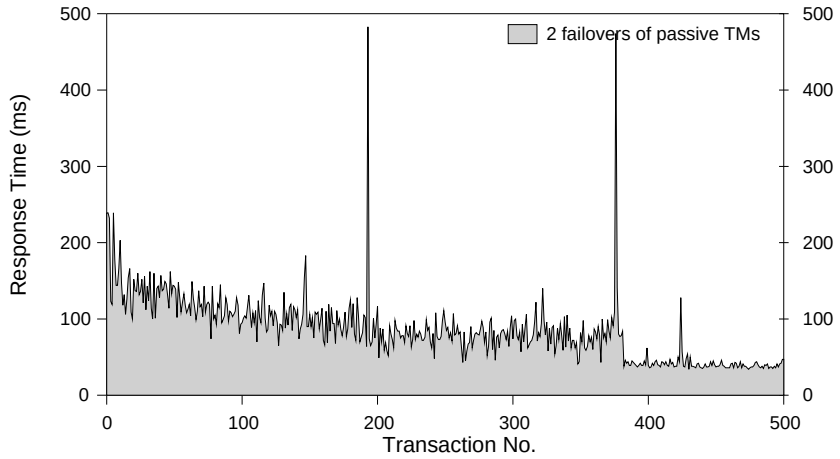


Figure 8.9: 2 failovers in a single testrun

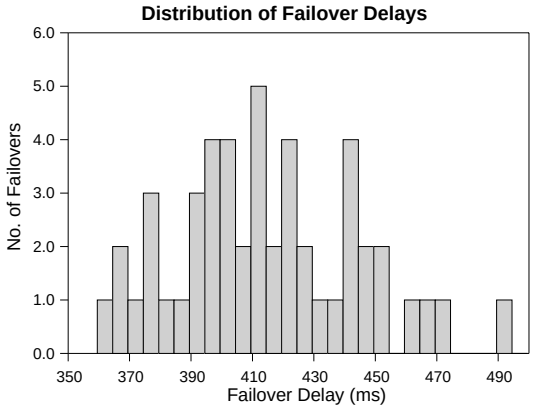


Figure 8.10: A histogram illustrating the distribution of failover-delays. Each bar has a horizontal size of 5 ms.

Response Time	`Connect-Exception`	`Socket-Exception`	`EOFException`
Minimum	3	40	306
Maximum	12	181	404
Average	8	102	362

Table 8.3: Response times in ms for various failure exceptions

A more detailed analysis of the transactions executed around the time of the failover was also performed. In particular, by logging the exceptions thrown because of a failure, the failover delay could be classified into three distinct classes:

- Connect failure. The client is unable to connect to the remote machine. This failure throws a `ConnectException`.
- Socket failure. The socket connection to the remote machine has not been torn down by the communication layer yet, but is torn down while the client is waiting for a reply. This failure causes a `SocketException`.
- EOF[2] failure. The remote machine fails while processing a request. It is interpreted as an unexpected EOF, and an `EOFException` is thrown.

The maximum, minimum and average measured response times for 50 testruns of these failures are listed in Table 8.3. Since the transaction participants are also clients of the transaction manager, multiple failovers can occur for a single transaction. For instance, if the primary fails while creating a transaction an EOF failure will occur at the client. After the failover the new primary creates the transaction. Then, both transaction participants try to join the transaction. However, they both have outdated information about the transaction manager primary. Thus, a Socket failure or a Connect failure will occur depending on the status of the old connection and a transaction might be delayed several times because of a single failover.

The EOF failure class causes the longest failover delays, and a closer inspection shows that the connection to the TM stays alive for an unnecessary long time. The client therefore observes a much larger failover delay than necessary.

An optimalization for getting more accurate results was needed. It exploits the fact that the failover time for a simpler application will be the same as for a more complex. This is because the underlying system (Jgroup/ARM) that manages the failover is the same. A 75 ms timeout of the `Socket` was added to the client, by using `setSOTimeout(75)`. Because the previous tests have high response times for the first transactions (over 150 ms), and the socket would timeout, a simpler application, called `FailoverServer` was introduced. It has only one method, which sleeps for 10 ms and then returns the name of the node.

The failover delay for the `FailoverServer` application, as observed from the client was between 200 and 250 ms for all 50 executed testruns.

[2]End-Of-File

Chapter 9

Discussion

This chapter summarizes and compares the test results gathered in Section 8.

9.1 Comparing the Test Results

Table 9.1 summarizes the results of the testruns made in Chapter 8. The response time average and standard deviation are presented[1]. It should be noted that these numbers only apply for these testruns and they should not be interpreted as any general response time guarantee, but rather as properties of the specific testrun. However, they can be used as a reference for comparisons between the individual testruns.

The following sections presents a summary of the results from the testruns and compares passive replication, active replication and the nonreplicated case. Finally, the failover delay is examined.

9.1.1 Passive Replication versus No Replication

Replication increases the overhead of a service. The results of Testruns 1–4, as presented in Table 9.1, 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 seems 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 nonreplicated case.

The standard deviation seems to change similarly to the average response time. It causes a significant leap when first replicated and then scales linearly when adding the third and fourth replica.

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

[1] Remember that the first 50 transactions of each testrun are disregarded in this discussion.

Testrun	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	189	27	302
7	3 passive TMs and 2x2 banks	236	40	402
Active replication of the TM				
8	2 active TMs and 2 banks	95	13	102
9	3 active TMs and 2 banks	100	12	113
10	4 active TMs and 2 banks	117	13	149

Table 9.1: A summary of the response times for the testruns in Chapter 8

Testrun	Response Time (ms)	Delay Rate (%)	Delay (ms)
M2	47	30	10–17
M3	62	60	12–20
M4	72	70	13–23
M5	49	35	10–20
M6	103	60	10–30
M7	119	65	10–36

Table 9.2: The overhead caused by group management

(Testrun 7), the response time is five times higher than in the nonreplicated case. This is a considerable amount, and a closer study was needed to figure out the cause of the delay.

First, the overhead caused by the group management threads is inspected. Then, the time to update the backups for various degrees of replication is studied. Finally, the results are combined and summarized.

Overhead Caused by Group Management

Testruns 2–7 were modified to see how much overhead the Jgroup/ARM system adds. The modified testruns (Testruns M2–M7) were designed to avoid multicasts to be able to see the overhead added by the group management threads. This was achieved by commenting out the code that sends the IGMI at the primary. The results are listed in Table 9.2. A more detailed study of each invocation reveals that some of the invocations are delayed. How often these delays occur and how large most of them are also shown in the table.

The results show that an increased degree of replication caused a higher rate of interruption and somewhat longer delays. This is caused by other threads in the Jgroup/ARM system interrupting the execution to handle group man-

Testrun	Unicasts	TM Multicasts	TP Multicasts
1	8	0	0
2	8	2 (x2)	0
3	8	2 (x3)	0
4	8	2 (x4)	0
5	8	0	2 (x2)
6	8	2 (x2)	2 (x2)
7	8	2 (x3)	2 (x2)

Table 9.3: The increased number of messages when replicating the TM

agement issues, and more replicas lead to an increase in the required group management facilities and the number of messages sent over the network.

Comparing the response time with the delay rate clearly shows that Testrun M7 has a delay rate that is too low for the response time. This is also apparent for Testrun M6, although to a smaller degree. The rest of the increase in the average response time is caused by larger delays for some of the invocations. Since the system is fully replicated, a single invocation might be delayed by many group management threads. Also, by spending more time in the system, it exposes itself to even more delays by the same reason.

These results indicates that just running the Jgroup/ARM system causes from 0–150 percent increase in the response time compared to the nonreplicated case, depending on the level of replication. Remember, however, that the results from Testrun 1 also include the time to persistently save the decision and outcome of each transaction as a part of 2PC. This is not true for Testruns M2–M7.

Messages per Transaction

The number of messages sent over the network increase with an increasing degree of replication. This is not only so for the group management issues mentioned above, but also for each transaction.

The backups are updated by an internal group method invocation (IGMI). IGMI uses multicast to invoke all members of the group including the replica where the message originated. To ensure that all members have been updated, an IGMI waits for all replicas to reply before it continues. The response time of an IGMI is therefore the same as the response time of the slowest replica.

Figure 9.1 shows all messages needed for a successful termination of a transaction in a fully, and passively, replicated system.

Table 9.3 presents the number of messages needed to successfully terminate a transaction from each of Testruns 1–7. The numbers in parentheses indicate the number of recipients for each multicast.

A transaction in a nonreplicated environment causes 8 unicasts and 8 replies for a total of 16 messages. This can be seen by counting the unicast messages in Figure 9.1. The pairwise parallel prepare and commit invocations are only counted as two. In comparison, for any fully replicated environment (Testruns 6 and 7) 4 serial multicasts, plus replies, are needed in addition to backup the transaction's prepared and committed state. The multicast messages, IGMI, are sent to both backups (shown as grey lines in Figure 9.1) and the primary.

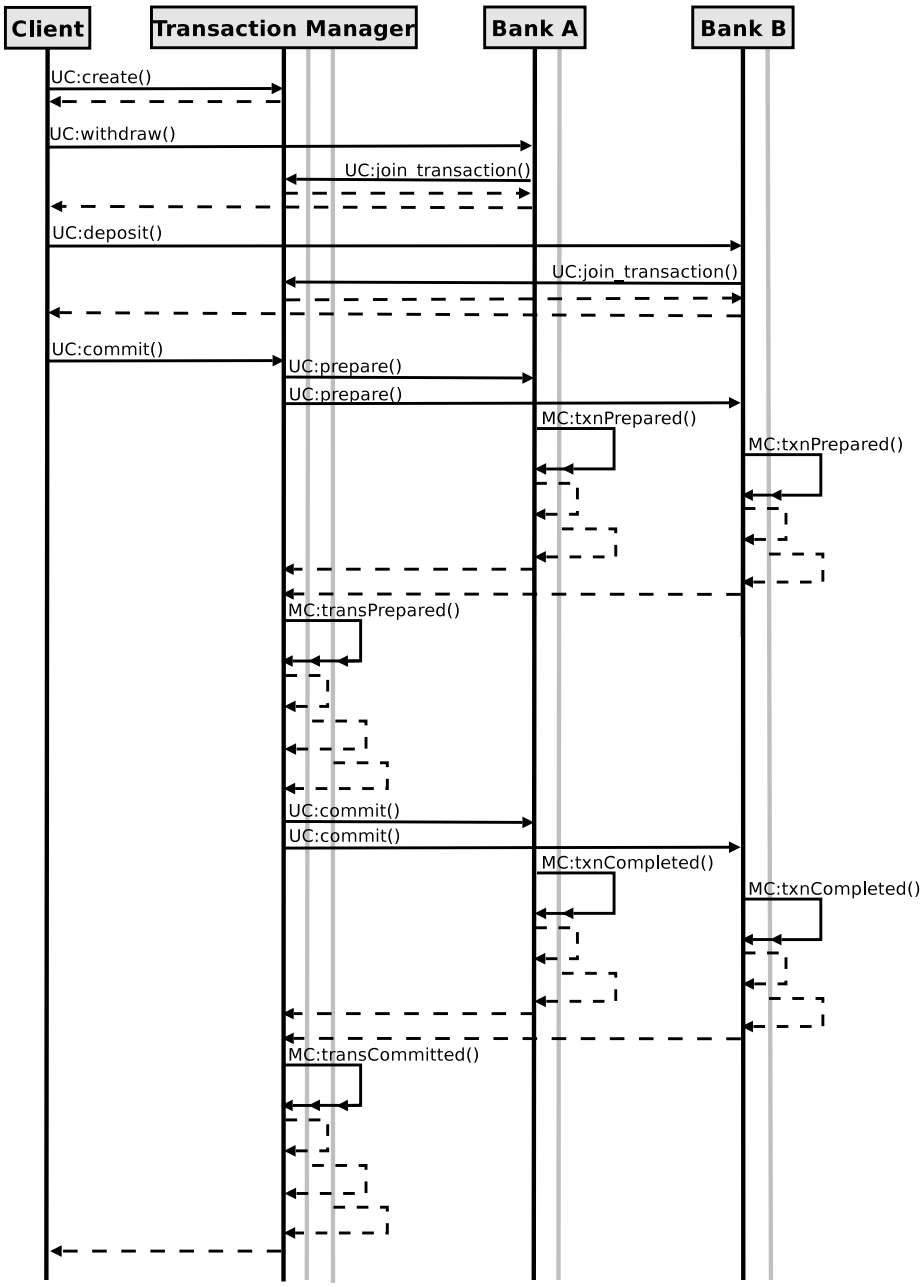


Figure 9.1: The messages in a fully replicated system. UC = unicast, MC = multicast. A nonreplicated system does not have the multicast messages. The number of unicasts is the same for a nonreplicated system as for a passively replicated one.

Testrun	Transaction Manager		Transaction Participant		Total
	Decision	EOT	Prepare	Commit	
2	20	10	0	0	30
3	22	11	0	0	33
4	24	12	0	0	35
5	0	0	12	13	25
6	49	11	12	13	85
7	94	12	13	13	132

Table 9.4: The cost of updating the backups

The transaction participants can perform their multicasts in parallel so only two multicasts are counted for all of them. However, the response time of a prepare or commit invocation from the TM is the response time of the slowest replica of all participants. As shown earlier in this section, increasing the number of recipients of a multicast increase the response time.

Time to Update the Backups

The response times in Table 9.2 (Testrun M2–M7) are lower than the response times in Table 9.1. The lacking pieces are the processes of updating the backups.

As explained in Section 7.2.3, the transaction manager must make the decision to commit and the mark of the end of transaction *durable*. This is done by an IGMI. Table 9.4 displays the cost of updating the backups for Testruns 2–7. The measurements are done at the primary over the invocation of each of the IGMI-methods in Figures 7.8 and 7.10. The methods are marked with MC (multicast) in Figure 9.1.

Testruns 2–4 shows a small, but steady increase in the time to backup the transaction. This is probably due to the fact that it waits for the slowest backup before it proceeds. For the same testruns the decision takes twice as long as the end-of-transaction (EOF) to backup. This is because more work needs to be done at each backup when receiving the first.

Testruns 5–7 show that the cost of updating the transaction participant backups is around 25 ms. It is slightly faster than updating the transaction manager backups. For Testruns 6 and 7, however, backing up the TM decision is slow. A closer look at the primary side reveals that the time is spent while the primary waits for the answer from the communication layer. On the backup-side the time is spent while reading the objects in the argument.

The same delay was observed for Testrun 2 if the client was set to wait 200 ms between each transaction. It seems that if there is too long time between two updates, a significant delay is added. This delay is probably caused by caching of objects by `java.io.ObjectOutputStream`. If the time between two invocations is too long, however, the caches are disregarded, causing increased response times.

The exact source of the delay has not been found, but it is caused by the unmarshaling of the `GroupTxnManagerTransaction` object that is sent as an argument to the `transPrepared()` method. When an object is received, the JVM checks if the class definitions of the arguments exist locally. If it does not, it must be built from the received object, and the referenced objects must also

Testrun	Messages	No Backup (ms)	Time To Backup (ms)	Total (ms)	Error	
					(ms)	(%)
1	8 UC	47	0	47	0	0
2	8 UC, 2(x2) MC	47	30	77	0	0
3	8 UC, 2(x3) MC	62	33	95	3	3
4	8 UC, 2(x4) MC	72	35	107	1	1
5	8 UC, 2(x2) MC	49	25	74	1	1
6	8 UC, 4(x2) MC	103	85	188	-1	1
7	8 UC, 2(x3) + 2(x2) MC	119	132	251	15	6

Table 9.5: A summary of the response times for the testruns in Chapter 8

be read from the stream, which takes time.

Summary

Table 9.5 summarizes the discussion of Section 9.1.1. The last two columns give the difference of the result when adding Testruns M2–M7 with the time to update the backups and the response time of the original testruns (Testruns 2–7). Testrun 1 has been added for easier comparison. Most of them are within a few percent off. This is due to minor variances in the response time. Testrun 7, however, has a deviation of 6 percent. This is probably due to a varying amount of network traffic for the switch to which the cluster is connected.

When adding more replicas, additional network traffic occurs, both the transaction execution and the group management demand more processor resources, and more garbage collection is needed. Thus, a transaction, which is run in a replicated environment, will be delayed by the threads for group management more often, causing a slower response time. Also, the effect unmarshaling has on the response time (as earlier in this section) leads to highly increasing response times if the transaction execution is delayed. Testruns 6 and 7 would have had a time to backup of about 45 and 52 ms, respectively, and a total response time of 143 and 171 ms, if the source of this delay had been found and removed. This would yield a total delay of 204 and 263 percent, which would be give much better performance results.

Unfortunately, comparable measurements on other systems have not been found in scientific articles. It is therefore hard to accurately evaluate the results of pGahalo.

9.1.2 Passive Replication versus Active Replication

The cost of active replication is explored in Testruns 8–10. The results are displayed in Table 9.1. Compared to Testrun 1, executing two actively replicated TMs (Testrun 8) doubles the response time and adds some variance. Adding further replicas increase the response time with about 10–20 percent for each, but leave the standard deviation unchanged.

A transaction successfully executing the test scenario using pGahalo (Testruns 2–4) causes two multicasts: One when backing up the commit decision, and another when all participants have acknowledged the commit. These are

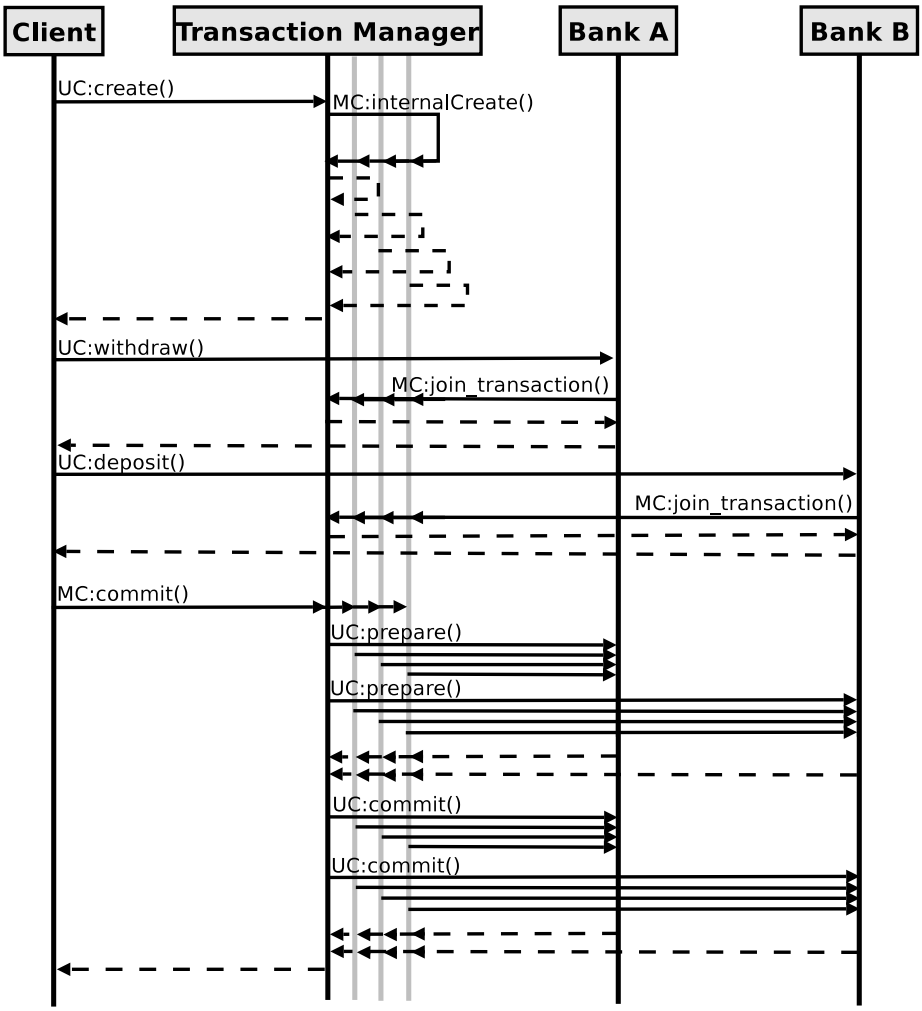


Figure 9.2: The messages in a system with an actively replicated transaction manager. UC = unicast, MC = multicast. Only the first reply for each EGMI multicast is shown.

Testrun	Unicasts	IGMI	EGMI
8	5	1 (x2)	3 (x2)
9	5	1 (x3)	3 (x3)
10	5	1 (x4)	3 (x4)

Table 9.6: The messages sent in Gahalo

shown in Figure 9.1. Remember that the participants are nonreplicated in this discussion. In Gahalo the same transaction (Testruns 8–10) causes four multicasts (see Figure 9.2): One IGMI to replicate the creation of a transaction[2], one EGMI for the join of each participant, and one EGMI for the commit invocation from the client. Since multicast is generally slower than unicast, pGahalo normally has a shorter response time than Gahalo, in spite of a being able to continue after receiving the first reply.

The results indicate that active replication gives slower, but more stable response times. The difference is, however, not very large. The main advantage of passive replication is therefore less resource usage, since the backups do not need to perform every task, but can be updated when necessary by the primary.

9.1.3 Failover Delay

The observed client-side failover delay for the transaction test was found to be as much as 360–490 ms (see Section 8.4). However, the failover 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 [GR93] distinguish five classes of transaction-oriented computing, with various properties and requirements. According to this classification the failover 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 failovers of less than 200 ms, depending on the application.

[2]The creation of a transaction is a nondeterministic operation. Thus, to keep consistent, only one replica of Gahalo creates the transaction and sends the result to the other group members.

Chapter 10

Conclusion and Further Work

This chapter summarizes the contributions of the thesis. In addition, some directions for further work are pointed out.

10.1 Conclusion

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. To be able to achieve both liveness and safety, however, the techniques should be integrated in such a way that non-determinism is handled.

This thesis 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 implementational basis of the combination, Jini and Jgroup/ARM, is given in Chapter 4, where the underlying systems are presented. Chapter 5 provides a detailed description of the challenges of nondeterministic execution in a replicated environment and existing solutions to the problems. The approach developed in this thesis is presented in Chapter 6. Together these chapters constitute a framework for integration of transactions and replication. The implementation of the integration is presented in detail in Chapter 7. The framework is based on allowing the transaction manager to break replication transparency.

The tests in Chapter 8 show that transactions can be executed in a passively replicated environment with a 300 percent increase in the response time. In addition, passive replication of the transaction manager was observed to be between 10 and 20 percent faster than active replication, but with a cost of higher variance. Also, a failure of the primary will cause a failover delay of about 400 ms on average for the transaction manager. Measurements on a smaller application, however, indicate that the real failover time is probably closer to 200 ms.

For a real world application the cost of replication must be weighed against the advantage of increased availability. If the system cannot tolerate the down-time 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 thesis 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 despite of failures.

10.2 Further Work

Several issues have been left for further research. These include:

- Complete error handling for the transaction manager should be developed. Once this is available, error insertions can be done and extensive measurements of the resulting response time can be made.
- The presumed abort two phase commit protocol should be optimized. This includes enabling read votes for the transaction participants and making the backup of the end-of-transaction asynchronous. Also, an early reply of the transaction outcome can be given to the client as soon as the transaction manager has persistently saved the decision. This would improve the response time for a transaction in a replicated environment by tens of milliseconds.
- On the client side, the time to close the socket to a failed server is too long. A mechanism to be able to detect such failures from the client side should be embedded in the communication layer. This would improve the failover delay observed by the clients.
- Mechanisms to ensure a complete fulfillment of the exactly-once execution semantics (e.g. retry of failed transactions) should be implemented. This ensures that all operations of each transaction are logically executed once. Also, explicit end-user interaction control should be added to guarantee that a user will perceive that each request generates exactly one reply.
- Currently, if a participant fails, the entire transaction is aborted. The possibility of using nested transactions to avoid a complete abortion, and subsequent retry, should be investigated. This could have major impact on the transaction response time in the presence of partial system failures.
- This work assumes that all servers are controlled by a single entity (e.g. company). If nodes are spread over a WAN or the Internet, there will be multiple entities involved. The entities may have different goals and requirements, thus, a transaction manager as the one explained may not be accepted by all of them. Other mechanisms will be required to ensure the consistency of such a system.

- The system in this thesis consists of only one primary transaction manager that controls the transaction execution. The possibility of using multiple sets of primary and backups for load balancing should be investigated. This would also lead to fewer transaction should a primary fail since only the transactions handled by the failed transaction will be aborted.
- The prototype can be extended to include a distributed concurrency control like Strict 2PL. This will ensure that transactions are correctly serialized.

Bibliography

[ALv] Apache License, version 2.0 (ALv2). http://www.apache.org/licenses/LICENSE-2.0.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *j-IPL*, 21:181–185, 1985.

[ASW+01] Ken Arnold, Robert Scheifler, Jim Waldo, Bryan O'Sullivan, and Ann Wollrath. *The Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., second edition, 2001.

[Ban98] Bela Ban. JavaGroups - group communication patterns in Java. Technical report, Department of Computer Science, Cornell University, April 1998.

[Bar01] Jose Barrera. What is Java reflection? *Java Developers Journal*, 6(9), September 2001.

[BCH+98] A. Baratloo, P. E. Chung, Y. H. Huang, S. Rangarajan, and S. Yajnik. Filterfresh: Hot replication of Java RMI server objects. In *Proceedings of the* 4^{th} *Conference on Object Oriented Technologies and Systems (COOTS)*, pages 59–63, Santa Fe, New Mexico, USA, 1998. USENIX.

[BHG86] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[Bir93] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.

[BJ87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, New York, NY, USA, 1987. ACM Press.

[BMST93] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Distributed systems. In S. Mullender, editor, *Distributed Systems*, ACM Press, chapter 8: The primary-backup approach, pages 199–216. Addison-Wesley, second edition, 1993.

[BT93] Özalp Babaoğlu and Sam Toueg. Understanding non-blocking atomic commitment. Technical Report UBLCS-93-2, Laboratory for Computer Science, University of Bologna, January 1993.

[CHY+98] P. Chung, Y. Huang, S. Yajnik, D. Liang, and J. Shih. Doors: Providing fault-tolerance for CORBA applications. In *Proc. of the IFIP International Conference on Distributed System Platforms and Open Distributed Processing (Middleware '98)*, September 1998.

[Con03] World Wide Web Consortium. *Simple Object Access Protocol*. World Wide Web Consortium, version 1.2 edition, June 2003.

[Coo85] Eric C. Cooper. Replicated distributed programs. In *Proceedings of the tenth ACM symposium on Operating systems principles*, pages 63–78. ACM Press, 1985.

[DG01] Eliezer Dekel and Gera Goft. ITRA: Inter-tier relationship architecture for end-to-end QoS, 2001.

[DGP04] Partha Dutta, Rachid Guerraoui, and Bastian Pochon. Fast non-blocking atomic commit: An inherent trade-off. *Inf. Process. Lett.*, 91(4):195–200, 2004.

[DS02] X. Défago and A. Schiper. Specification of replication techniques, semi-passive replication and lazy consensus. Technical Report IC/2002/007, École Polytechnique Fédérale de Lausanne, Switzerland, February 2002.

[DSS98] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 43–50, West Lafayette, IN, USA, October 1998.

[FG99] Svend Frølund and Rachid Guerraoui. Transactional exactly-once. Technical report, Hewlett-Packard Laboratories, July 1999.

[FGS98] Pascal Felber, Rachid Guerraoui, and Andre Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[FN02] Pascal Felber and Priya Narasimhan. 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*, pages 737–754. Springer-Verlag, 2002.

[GNSY00] A. Gokhale, B. Natarajan, D. C. Schmidt, and S. Yajnik. DOORS: Towards high-performance fault-tolerant CORBA. In *Proceedings of the* 2^{nd} *International Symposium on Distributed Objects and Applications (DOA '00)*, Antwerp, Belgium, 2000. Object Management Group.

[GNU] The gnu lesser general public license (lgpl). http://www.gnu.org/licenses/lgpl.html.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[HBH96] Abdelsalam A. Helal, Bharat K. Bhargava, and Abdelsalam A. Heddaya. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1996.

[Inc03a] Sun Microsystems Inc. *Jini Architecture Specification*. Sun Microsystems Inc., version 2.0 edition, June 2003.

[Inc03b] Sun Microsystems Inc. *Jini Technology Core Platform Specification*. Sun Microsystems Inc., version 2.0 edition, June 2003.

[JPPMAA01] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Sergio Arévalo. A low-latency non-blocking commit service. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 93–107. Springer-Verlag, 2001.

[KH04] Heine Kolltveit and Svein-Olaf Hvasshovd. Techniques for achieving exactly-once execution semantics and high availability for multi-tier applications. http://www.idi.ntnu.no/~kolltvei, 2004.

[LMW] Zhongwei Li, Ziangjiang Ma, and Yiwen Wang. Extension of rmi with group communication. Technical report, Cornell University.

[LS00] Mark C. Little and Santosh K. Shrivastava. Integrating group communication with transactions for implementing persistent replicated objects. *j-LECT-NOTES-COMP-SCI*, 1752:238–253?, 2000.

[Maf95] Silvano Maffeis. Adding group communication and fault-tolerance to CORBA. In *Proceedings of the USENIX Conference on Object-Oriented Technologies*, pages 135–146, Monterey, CA, June 1995.

[MDB01] Alberto Montresor, Renzo Davioli, and Özalp Babaoğlu. Jgroup: Enhancing Jini with group communication. In *Proceedings of the ICDCS Workshop on Applied Reliable Group Communication*, April 2001.

[MGG95a] K. R. Mazouni, B. Garbinato, and R. Guerraoui. Building reliable client-server software using actively replicated objects. In I. Graham, B. Magnusson, B. Meyer, and J.-M. Nerson, editors, *Proceedings of the TOOLS EUROPE'95 Conference*, pages 37–51, Versailles, France, 1995. Prentice-Hall.

[MGG95b] Karim R. Mazouni, Benoît Garbinato, and Rachid Guerraoui. Filtering duplicated invocations using symmetric proxies. In *Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems*, page 118. IEEE Computer Society, 1995.

[MH01] Hein Meling and Bjarne E. Helvik. ARM: Autonomous replication management in jgroup. In *Proceedings of the 4th Europen Research Seminar on Advances in Distributed Systems (ERSADS)*, Bertinoro, Italy, May 2001.

[MLO86] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, 1986.

[MMBH02] Hein Meling, Alberto Montresor, Özalp Babaoğlu, and Bjarne E. Helvik. Jgroup/ARM: A distributed object group platform with autonomous replication managmenet for dependable computing. Technical Report UBLCS-2002-12, University of Bologna, October 2002.

[Moh04] Parastoo Mohagheghi. *The Impact of Software Reuse and Incremental Development on the Quality of Large Systems.* PhD thesis, NTNU, Trondheim, Norway, July 2004.

[Mol04] Rohnny Moland. Replicated transactions in Jini. Master's thesis, University of Stavanger, July 2004.

[Mon00] Alberto Montresor. *System Support for Programming Object-Oriented Dependable Application in Partitionable Systems.* PhD thesis, University of Bologna, Italy, March 2000. Technical Report UBLCS-2000-10.

[MSEL99] G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little. Design and implementation of a CORBA fault-tolerant object group service. In *Proceedings of the 2nd IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 361–374, Helsinki, Finland, 1999.

[Nar99] P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, University of California, Santa Barbara, California, USA, September 1999.

[Nar01] Nitya Narasimhan. *Transparent Fault Tolerance for Java Remote Method Invocation.* PhD thesis, University of California, Santa Barbara, June 2001.

[New04] Jan Newmarch. Guide to Jini technologies. http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml, October 2004. Version 3.06.

[OMG04a] Inc. Object Management Group. *The Common Object Request Broker: Core Specification.* Object Management Group, Inc., version 3.0.3 edition, March 2004. formal/04-03-01.

[OMG04b] Object Managment Group. *Fault Tolerant CORBA*, March 2004. OMG Technical Committee Document formal/04-03-21.

[ÖV99] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems (2nd ed.)*. Prentice-Hall, Inc., 1999.

[PKS03a] S. Pleisch, A. Kupšys, and A. Schiper. Preventing orphan requests in the context of replicated invocation. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pages 119 – 128, Florence, Italy, October 2003. IEEE.

[PKS03b] S. Pleisch, A. Kupšys, and A. Schiper. Replicated invocations. Technical report, Swiss Federal Institute of Technology (EPFL), September 2003.

[PMJPA01] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Group transactions: An integrated approach to transactions and group communication. In *Workshop on Concurrency in Dependable Computing*, Newcastle Upon Tyne, United Kingdom, 2001.

[Pol93] Stefan Poledna. 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.

[RBC+03] Yansong (Jennifer) Ren, David E. Bakken, Tod Courtney, Michel Cukier, David A. Karr, Paul Rubel, Chetan Sabnis, William H. Sanders, Richard E. Schantz, and Mouna Seri. AQuA: An adaptive architecture that provides dependable distributed objects. *IEEE Trans. Comput.*, 52(1):31–50, 2003.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[Sch93] Fred B. Schneider. *Replication management using the state machine approach*, pages 169–197. ACM Press/Addison-Wesley Publishing Co., 1993.

[Ske81] Dale Skeen. Nonblocking commit protocols. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142. ACM Press, 1981.

[Som03] Frank Sommers. Call on extensible rmi: An introduction to JERI. Technical report, JavaWorld - Jiniology, December 2003.

[SR96] André Schiper and Michel Raynal. From group communication to transactions in distributed systems. *Commun. ACM*, 39(4):84–87, 1996.

[Sun99] Sun Microsystems Inc. *Java Remote Method Invocation Specification*. Sun Microsystems Inc., revision 1.7 Java 2 SDK, version 1.3 edition, December 1999.

[Sun05] Sun Microsystems Inc. *Java Computing home page*, March 2005. http://www.sun.com/java.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, 2001.

[VBB+91] P Veríssimo, P. Barrett, P. Bond, 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*, ESPRIT Research Reports, chapter 9, pages 211–266. Springer Verlag, nov 1991.

[ZMMS02] W. Zhao, L. E. Moser, and P.M. Melliar-Smith. Unification of replication and transaction processing in three-tier architectures. *22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 290–297, 2002.