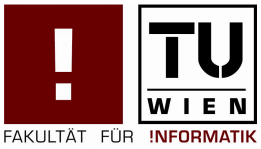

Master's Thesis

Transactional Replication in a .NET Environment

carried out at the

Information Systems Institute
Distributed Systems Group
Vienna University of Technology

under the guidance of

Dr. Karl Michael Göschka
and
Dr. Johannes Osrael
as the contributing advisor responsible

by

Juraj Oprsal, BSc.
Matr.Nr. 0125054
juraj.oprsal@centrum.sk

Vienna, 10th July 2008 _____

Acknowledgements

I would like to thank my advisor Johannes Osrael for his continuous support during the whole work. I also thank my supervisor Karl Michael Göschka for his assistance and valuable comments in the final phase of my thesis.

My further thanks go to my friends, especially Stanka and Evka, for providing special help to me.

Last but not least, I would like to thank my fiancée as well as my family for their extraordinary support throughout my studies at the university.

Abstract

Replication as an important mechanism in achieving dependability of distributed systems has already been subject to extensive research. Approaches to improve availability in degraded situations (e.g. network partitions) traditionally focus on pessimistic replication protocols, which prevent update conflicts during degradation. Recently, a novel class of optimistic replication protocols has been introduced, which are less restrictive than pessimistic protocols and allow trading consistency against availability. Tradeable and non-tradeable constraints can be defined in the system, while the former will be traded against availability during degradation.

The Active Replication Per Partition Protocol (ARPPP) is one concrete protocol applying this trade-off. It is a modification of the standard active replication scheme. In a healthy system, the ARPPP enforces strict constraint consistency and thus does not allow operations violating any of the constraints defined in the system. In case of node failures or network partitions, the ARPPP temporarily relaxes tradeable constraints and allows all operations not affecting non-tradeable constraints, even though tradeable constraints might become violated. After the network and all nodes recover from failures, strict constraint consistency must be reestablished in a process called reconciliation.

The contribution of this thesis is a prototype implementation and evaluation of the ARPPP in the Microsoft .NET environment using the C# programming language. Based on the features of the .NET Framework 2.0, the prototype provides full transaction support and allows nested replica calls as well.

Evaluation results indicate, that the additional necessity of constraint checking not only causes additional overhead and thus decreases performance, but in some cases also imposes higher transaction abort rates of concurrent operations. This is, however, expected behaviour, that admonishes to a careful constraint definition at application development time.

Zusammenfassung

Replikation als wichtiger Mechanismus in Erzielung von Zuverlässigkeit verteilter Systeme ist bereits umfangreicher Forschung unterlegt gewesen. Verschiedene Ansätze die Erreichbarkeit während Systemdegradation (z.B. Netzwerktrennung) zu verbessern haben sich traditionell auf pessimistische Replikationsprotokolle konzentriert, welche Update-konflikte während Degradation vermeiden. Vor kurzem wurde eine neue Klasse an optimistischen Replikationsprotokollen vorgestellt, die weniger restriktiv als pessimistische Protokolle ist und die es erlaubt, Erreichbarkeit und Konsistenz eines Systems auszubalancieren. Es können handelbare und nicht-handelbare Konsistenzbedingungen im System definiert werden, wobei die ersten während Systemdegradation gegen Erreichbarkeit gehandelt werden können.

Active Replication Per Partition Protocol (ARPPP) ist ein konkretes Replikationsprotokoll, das diese Balancierung anwendet. Es ist eine Modifikation der klassischen aktiven Replikation. In einem gesunden System fordert ARPPP strenge Konsistenz, also erlaubt keine Operation, die gegen eine beliebige Konsistenzbedingung stoßt. Im Falle eines Knotenausfalls oder einer Netzwerktrennung werden vorübergehend alle handelbaren Konsistenzbedingungen von ARPPP aufgehoben und somit alle Operationen, die keine nicht-handelbaren Bedingungen betreffen, erlaubt, auch wenn die handelbaren Bedingungen verletzt werden können. Nachdem das Netzwerk sowie alle Systemknoten heilen, wird strenge Konsistenz im System – während sogenannter Rekonciliation – wiederhergestellt.

Die Kontribution dieser Arbeit ist die Prototypimplementierung und –evaluierung des ARPPP Replikationsprotokolls unter Microsoft .NET in der C# Programmiersprache. Basierend auf Eigenschaften des .NET Frameworks 2.0, unterstützt der Prototyp sowohl Transaktionen, als auch geschachtelte Replikaaufrufe.

Die Ergebnisse der Evaluierung indizieren, dass der zusätzliche Bedarf an Überprüfen von Konsistenzbedingungen nicht nur einen höheren Aufwand mit sich bringt, sondern auch die Ursache für öftere Transaktionsabbrüche bei manchen gleichlaufenden Operationen ist. Dies ist jedoch das erwartete Verhalten, das auf den Bedarf einer sorgfältigen Definition von Konsistenzbedingungen in jeder Applikation verweist.

Contents

1 Problem Description

1.1 Motivation

Distributed systems [TS02] form an essential part of today's information systems. Resulting into efficient services and the accompanying economic profit they are massively evolving in many different fields in our information society. The increasing complexity of modern distributed systems evokes a higher need for their dependability [ALRL04].

Distributed object systems evolve aiming at specific user requirements. Failure transparency is one of the most widely required properties of a distributed system, using replication as the main method in achieving it.

Figures 1.1 and 1.2 show a scenario where the system needs to cope with a network split in order to ensure failure transparency:

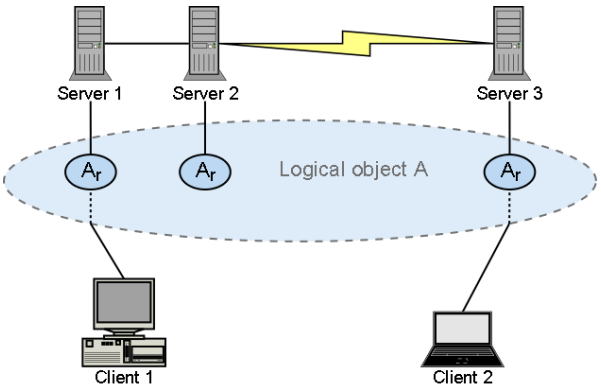

Figure 1.1: Example of a healthy distributed object system.

In a healthy system (figure 1.1), all read and write operations are allowed, while all changes to one object are replicated to other servers.

If the underlying network splits into several partitions (e.g. due to a link failure, see figure 1.2), object changes cannot be replicated to all servers anymore. In order to mask the network failure and to provide as much functionality as possible, distributed systems traditionally allow read operations and use one of two policies for write operations – they either *disable all* or *enable all* write operations.

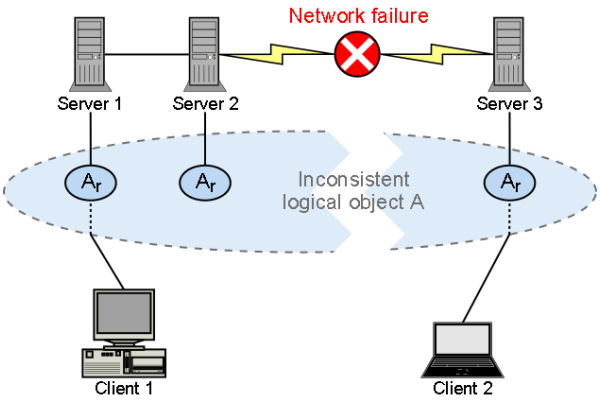

Figure 1.2: Distributed system partitioned due to a network failure.

In the first case the system availability obviously suffers. In the second case the system stays available, however replica inconsistencies between partitions may arise.

The DeDiSys project (http://www.dedisys.org) represents an innovative attempt to combine these extreme policies and introduces a new policy with adjustable trade-off between availability and consistency.

1.2 DeDiSys Project

DeDiSys (Dependable Distributed Systems) was a research project supported by the European Community with the main objective to develop a concept for balancing dependability in distributed systems by relaxing system consistency. The project investigated data-centric systems (e.g. distributed object systems), as well as service-centric systems (e.g. Webservice systems). Validation, comparison, and assessment of the developed models in combination with existing middleware technologies like EJB, CORBA, or Microsoft .NET was also an important objective of this project.

In the case of tightly coupled data-centric systems, DeDiSys introduced a system architecture with adaptable dependability based on a novel constraint consistency checking approach [OFG+06].

1.3 Problem Definition and Contribution

As already mentioned, traditional distributed systems will handle a network split using one of two extreme strategies, however both dealing with major disadvantages. Within the frame of the DeDiSys project several new replication protocols have been designed that combine these two strategies. The resulting replication model is called *Availability/Consistency Balancing Replication Model (ACBRM)* [OFG07]. Osrael [Osr07] discusses replication techniques for balancing data integrity with availability in detail in his PhD thesis.

The *Primary-per-Partition Protocol (P4)* [BBG+06] and *Adaptive Voting (AV)* [OFGG06] are two concrete protocols of the ACBRM based on the traditional Primary-Backup and Voting replication techniques, respectively. .NET-based prototype implementations of both protocols exist and are closely described in [Wei06, Hab06, Chl07].

In his PhD thesis [Osr07] mentions that Active Replication can also be adapted to adhere to the ACBRM. However, besides an informal description of the so-called Active Replication Per Partition Protocol by Osrael, neither a detailed specification of the protocol has been published nor an implementation exists. Therefore, the main contributions of this thesis are:

- A prototype implementation of the ARPPP in the .NET environment
- Performance evaluation of the prototype.

Existing prototype implementations of the P4 and AV based on the Microsoft .NET Framework lack the support for transactions. This makes it impossible to evaluate any concurrency behaviour of the protocols. To allow such an evaluation, full transaction support (including distributed and nested transactions) should be provided by the developed implementation of the ARPPP.

Also, previous implementations do not support references between replica objects. This is, however, necessary for the performance measurement of nested invocations. Therefore, the support of inter-replica references should be provided by this prototype implementation.

Another challenge to face is the combination of replication and transactions using the relatively new .NET Framework 2.0 transaction model. Together with the support of inter-replica references, performance of concurrent and nested invocations in the ACBRM replication model can be evaluated for the first time in the .NET environment.

1.4 Organization of this thesis

In section 2 a general overview of the three main replication models is provided. Followingly, the novel approach of adaptive dependability in replication is introduced. In the end, the main concepts regarding transactions are briefly described and an overview of the transaction mechanisms in the .NET Framework 2.0 is given.

Section 3 starts with presenting the ARPPP protocol to be implemented. Afterwards, the component architecture as well as design decisions regarding all fundamental components and services are presented. After description of the replica object design and implementation, the following mechanisms are being examined – *Transaction Support*, *Group Targeted Communication*, and *Replication and Call Processing*. For each mechanism, the implementation of the core components is described with respect to the overall architecture. Where relevant, considerations on performance are discussed.

Evaluation results of the prototype are summarized in section 4, including an interpretation of these results with respect to the implementation.

The last section concludes this work by summarising its results and outlines the possible course of the future work. Also, a summary of existing related work is given.

2 Replication Protocols and Transactions

2.1 Replication Techniques

Replication is an important technique in achieving reliability in a distributed system. Apart from distributed systems, replication has been also extensively studied in the area of database systems (in order to achieve higher performance). Solutions from both areas are similar, however, as [WPS+00] suggests, they may differ in several aspects like coordination mechanisms, assumptions, or provided guarantees. [WPS+00] introduces an abstract replication model, that will be used in this thesis not only to compare existing replication mechanisms, but also to describe the own solution developed for ARPPP (see section 3.1). This is being done with the aim to highlight the similarities and differences of replication mechanisms discussed in this thesis.

The abstract model defines five generic phases of the replication process:

1. **Request (RE)**
   Submission of the client's operation to one or more replicas.

2. **Server Coordination (SC)**
   Synchronization of the operation execution among all servers with the aim to achieve the same execution order of concurrent operations. This phase is usually implemented either by using *atomic broadcast (ABCAST)* [DSU04, Lam78], or by using *distributed locking* [WPS+00].

3. **Execution (EX)**
   The execution of the operation (on one or more servers, depending on the replication model).

4. **Agreement Coordination (AC)**
   Agreement on the operation result among all involved servers. In this phase either the use of the *two-phase-commit protocol (2PC)*[1], or the use of *view synchronous broadcast (VSCAST)* [BJ87, GS97] is made.

[1] discussed in section 2.3.2

5. Response (END)
Sending the operation result back to the client.

For concrete replication protocols some phases may be skipped, merged, reordered, or it may be iterated over them.

So far, many different replication protocols have been described, however the basic models (from which many others are derived) can be reduced to the following three: *Primary-Backup Replication*, *Quorum Consensus Replication*, and *Active Replication*.

2.1.1 Primary-Backup Replication

In Primary-Backup Replication [BMST93], also known as Passive Replication, the client sends the operation request to one replica server, called the *primary*. The primary then executes the operation and propagates the state update to all other servers, called *backups*. Two strategies are applied depending on the operation result that the primary sends back to the client. For *synchronous* (or *eager*) primary-backup replication, the result is sent after receiving acknowledgements about a successful state update from all backups. For *asynchronous* (or *lazy*) primary-backup replication, the result is sent immediately after operation execution. In this case, the state update propagation occurs (lazily) afterwards.

Both approaches, synchronous and asynchronous, have their benefits as well as drawbacks. The former, obviously, ensures replica consistency among all servers all the time, however increases transaction response times by incorporating the state updates in the transaction. This approach is used mainly in tightly coupled distributed systems. The latter does not guarantee replica consistency, but may be significantly faster, if not the only possible solution especially in loosely coupled environments, like mobile applications.

Primary-backup replication maps to the phases of the abstract model as follows:

1. Client sends the operation request to the primary replica server.

2. Server coordination phase (SC) is not necessary, because all clients send their requests to the primary and thus the request delivery order equals the execution order.

3. Primary replica server executes the operation.

4. Agreement Coordination (AC) is accomplished by propagating the state update from the primary to the backups. This can be done by VSCAST (mostly in distributed systems), or by coordination using the 2PC protocol (mostly in database systems). Both approaches provide correct behaviour even in case the primary fails [WPS+00, GS97].

5. Primary sends the operation result back to the client.

The phase order just described corresponds to synchronous (eager) primary-backup replication. In asynchronous (lazy) primary-backup, the last phase precedes agreement coordination. In this case the update propagation does not need to fulfill such strong requirements. More on asynchronous replication can be found in [GHOS96, GA87].

2.1.2 Quorum Consensus Replication

In Quorum Consensus (Voting) Replication [Tho79, Gif79] the replica servers "take a vote" on each operation. All servers are equivalent and thus all are allowed to handle operation requests. In order to perform an operation, the server needs to find a quorum consensus among sufficiently many replica servers. The minimum quorum sizes are chosen so, that no two conflicting operations can gather enough votes to execute concurrently. Assuming a simple case, that each server has exactly one vote, the following conditions must be satisfied for the sizes of the read quorum (rq), the write quorum (wq), and the whole system (N):

$$rq + wq > N \tag{1}$$

$$wq > N/2 \tag{2}$$

Condition 1 eliminates *read-write* conflicts, while condition 2 eliminates *write-write* conflicts.

The voting (quorum) systems can be modeled in a variety of ways as described in [Tho79, Gif79, PL88, PL91, JM87].

Compared to primary-backup replication, the risk of expensive coordination in the case of primary failure is eliminated. Furthermore, the quorum sizes can be adjusted on the properties of a concrete distributed system (like read/write operation ratio) and improve the overall performance. On the other hand, replica servers need some kind of synchronization to negotiate an order of the executed operations. This is accomplished implicitly by the acquirement of a quorum consensus [DGS85].

Depending on whether the quorum consensus is found before or after the operation execution, it can be distinguished between *pessimistic* and *optimistic* approaches [DGS85, SS05], respectively.

Also, *static* and *dynamic* [DB85, PL88] voting protocols are distinguished, depending on the determination of quorum sizes. For the former class, the quorum sizes are fixed, while for the latter, quorum sizes are dynamically adjusted according to some conditions (in most cases, according to the current server count).

The mapping of quorum consensus replication to the phases of the abstract model is as follows:

1. Client sends the operation request to any replica server.
2. The server then coordinates (SC) with others by looking for a quorum consensus on the requested operation. Distributed locking [WPS+00] is a widely used mechanism to achieve this.
3. Replica server executes the operation.
4. The server sends a state update to all servers in the current quorum and starts a coordination round (AC), e.g. 2PC, to gain each server's agreement on a successful operation finish.
5. Replica server sends the operation result back to the client.

Described above is the pessimistic approach. If optimistic, the task from phase SC is (implicitly) accomplished by the phase AC with the hope a quorum consensus will be found.

2.1.3 Active Replication

In Active Replication [WPS+00, Sch93], also called state machine replication, the operation request is broadcasted to all replica servers, which then execute the operation independently and send the result back to the client.

In this approach, deterministic behaviour of the operations is required. Otherwise, servers in a consistent state could end up, after executing the same operation, in an inconsistent state. To achieve determinism, replicas need to act as state-machines [Sch93]. Therefore, multi-threaded execution or the use of randomized operations are not allowed.

This determinism restriction is the main disadvantage in active replication. An advantage, on the other hand, is the simplicity of this technique. Also, in some implementations failure transparency regarding the client is easily achieved, since if a replica server fails, other servers will still process the operation[2].

The abstract model adapted to active replication looks like the following:

1. Client broadcasts the operation request to all replica servers.
2. Synchronization of concurrent operations (SC) is ensured either by ABCAST of the requests (actually in phase RE), or by using distributed locking.
3. All replica servers execute the operation.
4. If distributed locking is used in phase SC, all replica servers must commit the transaction by running a distributed commit protocol, usually the 2PC protocol.
5. Replica servers send the operation result back to the client. Depending on the implementation, the client can wait for all responses, for the majority of responses, or for the first response only.

[2] This can be a problem with implementations using transactions and the 2PC protocol, because if once a resource becomes involved in a transaction it usually must commit at the end. But if the resource becomes unavailable (because its server fails), the resource is not able to commit and the whole transaction will rollback (more on this issue in section 2.3.2).

As already sketched, transactions may imply some difficulties. Because a distributed transaction using 2PC must be committed by all resources involved, failure of a replica server will cause abortion of all running transactions. A possible solution how to circumvent this problem is by using another (non-blocking) commit protocol [GL06] (more on this issue in section 2.3.2).

If nested operations are allowed, the so called *redundant nested invocation (RNI)* problem [LFT02] must be handled. Figure 2.1 illustrates how the problem arises in a system with two server groups, each having two servers (separate server groups are used solely for the sake of lucidity).

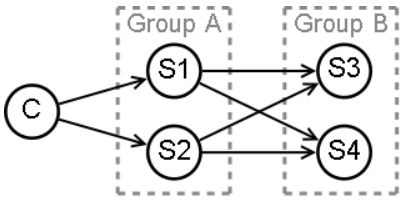

Figure 2.1: The redundant nested invocation (RNI) problem.

The client submits an operation to the group A and both servers (S1 and S2) begin to process it. The operation then triggers a sub-operation to the server group B. Because this happens on S1 as well as on S2, servers in group B receive the request twice. Such behaviour is redundant and thus causes unnecessary network load. [LFT02] describes the RNI problem as well as a particular solution.

Besides this, there exist several innovative approaches to active replication. For example, [JPA00] describes a solution enabling multi-threaded replica execution by introducing a deterministic thread scheduler. [FS01, Pec06, LXML07] are other examples of interesting solutions improving different aspects of this replication technique.

2.2 Replication with Adaptive Dependability

2.2.1 The Concept

The concept described here has been developed in the frame of the DeDiSys project (see section 1.2). The project identifies applications [JSSG05, OFKG05], where on the one hand, strict consistency is not desired all the time, but on the other hand, arbitrary inconsistencies are not acceptable either.

[OFKG05] discusses possible correctness criteria in data-, service-, and resource-centric distributed systems. For data-centric systems, constraint consistency (data integrity) is the investigated correctness criterion. The goal of the project is a replication model, where

> *"data integrity can be temporarily relaxed in order to enhance availability, i.e. constraint consistency can be traded against availability."* [OFG+06]

2.2.2 Terminology

To enable a description of the replication model applying this trade-off, some definitions (according to [OFG07]) need to be clarified:

Tradeable vs. non-tradeable constraints: *Non-tradeable constraints* must never be violated and thus cannot be traded for higher availability. *Tradeable constraints* can be temporarily violated (relaxed) in order to achieve higher system availability.

Critical vs. non-critical operations': A *critical operation* affects at least one non-tradeable constraint. An operation, which does not affect any non-tradeable constraint (but may affect any number of tradeable constraints) is called *non-critical operation.*

System modes (normal, degraded, reconciliation): The system resides in *normal mode* if all servers are reachable. In normal mode, the system guarantees strict constraint consistency.
If a server fails or the underlying network splits (e.g. due to a link

failure), the system switches to *degraded mode*. In this case, in order to achieve higher availability, tradeable constraints can be temporarily relaxed (i.e. non-critical operations are allowed).
After server restart, or a network re-unification, the system switches to *reconciliation mode* and establishes coarse replica consistency. In case all servers are reconciling, constraint consistency must be reestablished too (because switching to normal mode afterwards).
A state diagram modeling the described system modes and transitions between them is illustrated in figure 2.2.

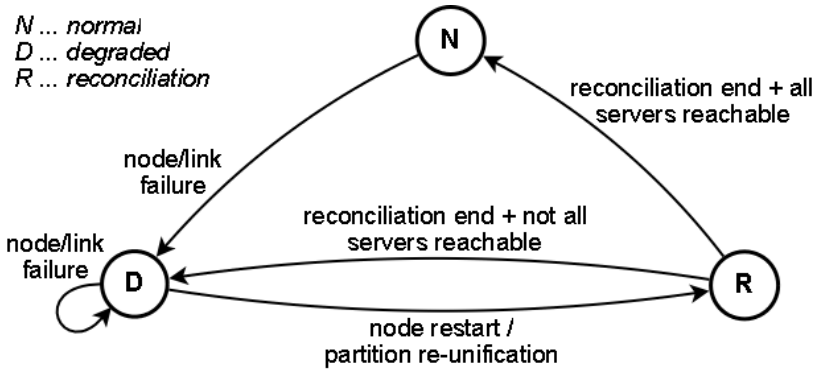

Figure 2.2: System modes and transitions between them.

2.2.3 Availability/Consistency Balancing Replication Model

A general replication model using this approach has been introduced in [OFG07] and is called the *Availability/Consistency Balancing Replication Model (ACBRM)*. Also, a system architecture implementing the model has been proposed in [OFG+06]. So far, two concrete replication protocols utilizing this approach have been described and implemented – the *Primary-per-Partition Protocol (P4)* [BBG+06] and *Adaptive Voting (AV)* [Chl07].

To enlighten things even more, [OFG07] extends the abstract replication model described in section 2.1, in order to conform to the ACBRM. For this purpose, additional phases in each system mode have been introduced.

In **normal mode**, only one new phase has been added, the *Constraint Validation (CV)* phase. It is inserted just after the EX phase, as can be seen in figure 2.3.

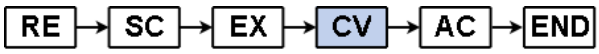

Figure 2.3: Phases of the ACBRM in *normal* mode.

The task of CV is to check, whether the result of the preceding execution (EX) fulfills all (tradeable and non-tradeable) constraints and in case of any constraint violation, aborts the operation. Of course, this makes sense only for write operations. Therefore, read operations simply skip this phase.

Since the concept of ACBRM tries to improve system availability in case of failures, the crucial extension is done in **degraded mode**. For this purpose, three phases have been added to the abstract model, as shown in figure 2.4.

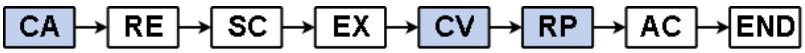

Figure 2.4: Phases of the ACBRM in *degraded* mode.

At the very beginning, the optional phase *Configuration Adjustment (CA)* is inserted. In this phase, some protocols may need to reconfigure some internal aspects of their behaviour. For example, a dynamic voting protocol might need to adjust quorum sizes, or a primary-backup protocol might need to redefine the primary replica in a partitioned environment. However, for some protocols this phase might be needless (e.g. static voting).
Similarly to normal mode, *Constraint Validation (CV)* is inserted after the EX phase, however, its task is modified. While critical operations are not allowed in degraded mode (critical constraints could be violated between partitions otherwise), non-critical operations are allowed. Evenmore, as tradeable constraints may be violated between partitions, there is not much sense to check them within one partition. Therefore, the protocol may decide simply not to check the tradeable constraints at all.
The last phase added (right after CV) in degraded mode is the *Reconciliation Preparation (RP)* phase. In this phase, the protocol logs information on the executed operations to prepare for later reconciliation. Different types and amounts of information [OFG07] can be logged, depending on the reconciliation strategy used.

Phases CV and RP in degraded mode have only relevance for write operations.

When partitions rejoin, the system switches to **reconciliation mode**. For this mode, two or, as the case may be, three phases have been added to the abstract model (see figure 2.5).

Figure 2.5: Phases of the ACBRM in *reconciliation* mode.

First, replica consistency, and eventually also constraint consistency must be established. These phases are called *Replica Consistency Re-establishment (RCR)* and *Constraint Consistency Re-establishment (CCR)*, respectively. If all replica servers are rejoining, both phases (RCR and CCR) must be performed. Else, if only a subset of servers is rejoining, it is sufficient to perform only RCR. However, CCR can be performed as well. Different strategies for CCR (*re-schedule and replay*, *stepwise rollback*, and *compensation actions*) are discussed more closely in [OFG07].
Because some protocols might need to adjust their behaviour after partition re-unification, an optional *Configuration Adjustment (CA)* phase is inserted.

2.3 Transactions

Transactions in programming represent a standard mechanism for synthesizing small atomic actions into complex operations, while ensuring some predefined properties. Although there exist several transaction concepts [GR92, Mos81, WS92], the term transaction usually refers to *ACID transactions* [GR92] (also in this thesis) characterized by the four properties – *Atomicity*, *Consistency*, *Isolation*, and *Durability*.

Atomicity Requires atomic behaviour of the transaction. Either the whole transaction is performed, or it has no effects at all. Atomicity is achieved by logging all changes made by a transaction and by applying or discarding them at transaction commit or rollback, respectively. For this purpose a *distributed commit protocol* must be used in a distributed system (see section 2.3.2).

Consistency If starting from a consistent system state, after executing a transaction, the system must end up in a consistent state, again. To provide this, the transaction is simply not allowed to commit until all system specific constraints are satisfied.

Isolation Changes made by a running (not yet committed) transaction should be visible neither to the system, nor to other concurrent transactions. What more, during execution of any two transactions, concurrent access to resources must be controlled in order to prevent interference between these transactions (see section 2.3.1).
The requirements just described refer to the so called *serializable* isolation. The ANSI/ISO SQL-92 specification also defines three other isolation levels with weaker requirements. These weaker isolation levels provide better performance, but can cause several phenomena like Dirty Reads, Non-repeatable Reads, or Phantoms [BBG+95].

Durability After a transaction commit, all changes made must unconditionally become durable and must be visible to the system (e.g. be written to the disk).

2.3.1 Concurrency Control

To provide concurrency consistency, the solution is to make transactions *serializable* [BHG87]. This means, the execution of any two concurrent transactions must be equivalent to a serial execution of these transactions.

To achieve serializability, transaction operations must be properly *scheduled* [BHG87], i.e. ordered in such a way serializability is secured. Proper scheduling may be done by several techniques. [BHG87] discusses *two-phase locking (2PL)*, *non-locking techniques* (like timestamp ordering, serialization graph testing, or certifiers), and *multiversion concurrency control*. From all these techniques, 2PL is the most widely used one.

Locking is a standard mechanism in synchronizing access to a shared resource. The 2PL technique divides the transaction process into two phases, which will be called here the *lock acquisition (LA)* and the *lock release (LR)* phase. The most important rule is that no lock can be acquired after any lock has been released. Usually, transactions release all locks together at the transaction commit or rollback. This is denoted as *Strict 2PL*.

In general, 2PL is a subject to *deadlocks*. A simple deadlock situation arises for example, if transaction A waits for a lock already owned by another transaction B and B, at the same time, waits for a lock owned by A. Of course, more complex deadlock situations are possible. To handle deadlocks, different mechanisms can be implemented [BHG87, BG81]:

Prevention The most simple technique is to not allow any transaction to be waiting. If a transaction cannot acquire a lock, either this transaction or the transaction owning the lock is aborted. Another simple technique is timeout. If a transaction waits too long for a lock, it is supposed to be deadlocked and thus is aborted.

Detection Another technique to handle deadlocks is to detect them precisely. This requires maintenance of a so called *waits-for graph (WFG)* keeping track of waits-for dependencies between transactions. Any deadlock situation appears as a cycle in WFG. In such case, one transaction from the cycle is aborted, for what exist different selection methods.

Avoidance The last solution to handle deadlocks is to avoid them completely. To bring an example, *Conservative 2PL* avoids deadlocks by requiring that all locks used during a transaction must be acquired before the first operation is executed.

For distributed transactions, concurrency control is usually distributed too. Several solutions for distributed locking have been proposed – *Distributed 2PL* [BHG87], *Wound-Wait (WW)* [RSL78], *Basic Timestamp Ordering (BTO)* [BG81], or *Distributed Certification* [SNM85]. All these solutions work in combination with the two-phase commit (2PC) distributed commit protocol (see section 2.3.2), which guarantees atomicity of a transaction.

Again, distributed 2PL is the most often used one and thus will be examined in more detail. [BHG87] shows that if Strict 2PL is used for the distributed variant, no communication between schedulers on different servers is required to ensure the 2PL rules. More complicated is the handling of distributed deadlocks. [BHG87] discusses both distributed deadlock prevention (by using timeouts or timestamps) and distributed deadlock detection (by using *global deadlock detection* or *distributed cycle detection*).

Global deadlock detection is accomplished by unifying local WFGs into a global WFG. A problem arises from the graph unification, which happens only periodically and thus, the deadlock detection must happen periodically too. For this reason, the discovery of a deadlock can be delayed, which blocks resources. Also, because the global WFG is not always up to date, stale transaction dependencies may cause so-called *phantom deadlocks*.

A distributed cycle detection algorithm called *path pushing* [BHG87] is based on the exchange of partial information from local WFGs with other servers. Under some circumstances, this approach discovers deadlocks faster than the global deadlock detection.

2.3.2 Distributed Commit

When a distributed transaction receives a commit request a distributed commit protocol must be initiated in order to ensure atomicity of the whole transaction. Most of the protocols are variants of a centralized *Two-Phase Commit (2PC)* protocol [GL06].

In 2PC, after receiving the commit request, the coordinator first requests all participants to *prepare* for commit. After receiving the answers, the coordinator decides whether the transaction can be committed or not. Depending on the decision, the coordinator then sends a *commit* or *abort* command to each participant. A disadvantage of the 2PC is, that in case of the coordinator failure the protocol execution can become blocked until the coordinator recovers.

A solution to the blocking problem of 2PC are non-blocking protocols. For example, [Ske81] introduces the *Three-Phase Commit (3PC)* protocol able to overcome blocking in some cases, however, for the price of an additional coordination phase. [GHR97] presents an optimistic commit protocol called *OPT*, which allows a transaction to "lend" a lock from a transaction that is already in the prepare phase. A different approach to handle server failures has been proposed in [GL06]. Here, the *Paxos Commit* uses the well known Paxos consensus algorithm to reach a commit or abort decision.

[GHR97] describes or lists several other variants of the 2PC and other commit protocols.

2.3.3 .NET Transaction Specifics

The ARPPP implementation to be developed in the frame of this thesis should reside on top of the Microsoft's .NET Framework 2.0. Because the transaction integration is a substantial goal, an overview of transaction support in the .NET Framework is given at this place.

The .NET Framework 2.0 provides a completely new transaction support mechanisms compared to its previous versions. [Low05] describes the main features of the new `System.Transactions` namespace and [Laz08] presents several articles on transactions under Microsoft Windows and in the .NET Framework.

One of the main improvements is the introduction of the *Lightweight Transaction Manager (LTM)* that provides better performance for simple transactions when compared to the already known *Distributed Transaction Coordinator (DTC)*. The LTM is used for transactions involving at most one durable resource (e.g. a database) and any number of volatile resources (see below), unless they reside in different application domains. If a second durable resource or any resource from a new application domain gets involved, the transaction is *promoted* to a distributed one and the DTC is used. If a transaction is serialized, the promotion is forced, because this may imply the distribution to another application domain.

Also, the `System.Transactions` namespace simplifies implementation of custom resources able to enlist in a transaction, the so-called transactional *resource managers (RM)*. To provide flexibility, two types of RMs can be defined:

Durable RMs store the data and the transaction history to a persistent storage and in case of failure they are able to recover and eventually continue in the transaction. An example of such RM is a database manager.

Volatile RMs manage only volatile data and do not need to recover after failures. An example could be an in-memory data structure managed by a transactional RM.

The DTC implements the 2PC protocol in order to ensure atomicity of distributed transactions. To correctly interact with custom RMs, every RM must implement the `IEnlistmentNotification` interface (see figure 2.6).

Another simplification represents the concept of *ambient transactions*. Whenever a transaction is started, it is stored in a *threadstatic* field `Current` of class `Transaction`, i.e. a static field within the current thread and the current application domain. Therefore, any method called on this thread (and within this application domain) can find out whether it resides in the scope of a transaction. Software components able to discover the ambient transaction and enlist with it as a RM are called *transaction-aware* components.

Transactions can be incorporated into a program by either *declarative* or *explicit* transactional programming model. The declarative model is the same one as in previous versions of .NET, however the transaction management behind it makes use of improvements introduced with the LTM whenever possible. The explicit model allows to completely manage the transaction flow by the programmer while offering two approaches. The easier one is to use the `TransactionScope` class as shown in listing 2.1.

```
using(TransactionScope ts = new TransactionScope())
{
  /* Perform any transactional work here... */
  ts.Complete();
}
```

Listing 2.1: Use of the `TransactionScope` class

In this case, creating the `TransactionScope` does all the work. To compare it with the second approach, listing 2.2 shows a code snippet doing exactly the same thing, however using plain transaction objects.

```
Transaction oldAmbient = Transaction.Current;
CommittableTransaction committableTransaction;
committableTransaction = oldAmbient as CommittableTransaction;
if(committableTransaction == null)
{
   committableTransaction = new CommittableTransaction();
```

```
   Transaction.Current = committableTransaction;
}

try
{
   /* Perform any transactional work here... */
   committableTransaction.Commit();
}
finally
{
   committableTransaction.Dispose();
   //Restore the ambient transaction
   Transaction.Current = oldAmbient;
}
```

Listing 2.2: Use of plain transaction objects

Unless using the plain transaction objects cannot be avoided, `TransactionScope` should be the preferred way to manage transactions. However, scenarios, where using a `TransactionScope` is not possible, exist. For example, if the transaction must be started within one method, but needs to be finished within another method, using a `TransactionScope` is not suitable.

When programing with the plain transaction objects, three classes are to be considered – `Transaction`, `CommittableTransaction`, and `DependentTransaction` as illustrated in figure 2.6.

`CommittableTransaction` is always the "top-most" `Transaction` object created and is also the only one that can be committed. If a nested transaction is needed, the `Transaction.DependentClone` method can be called to create one, called here a `DependentTransaction`. It is not possible to commit a `DependentTransaction`. However, calling its `Complete` method indicates to the parent all work has been done.

The `CommittableTransaction` with all `DependentTransactions` cloned from it belong to the same logical transaction.

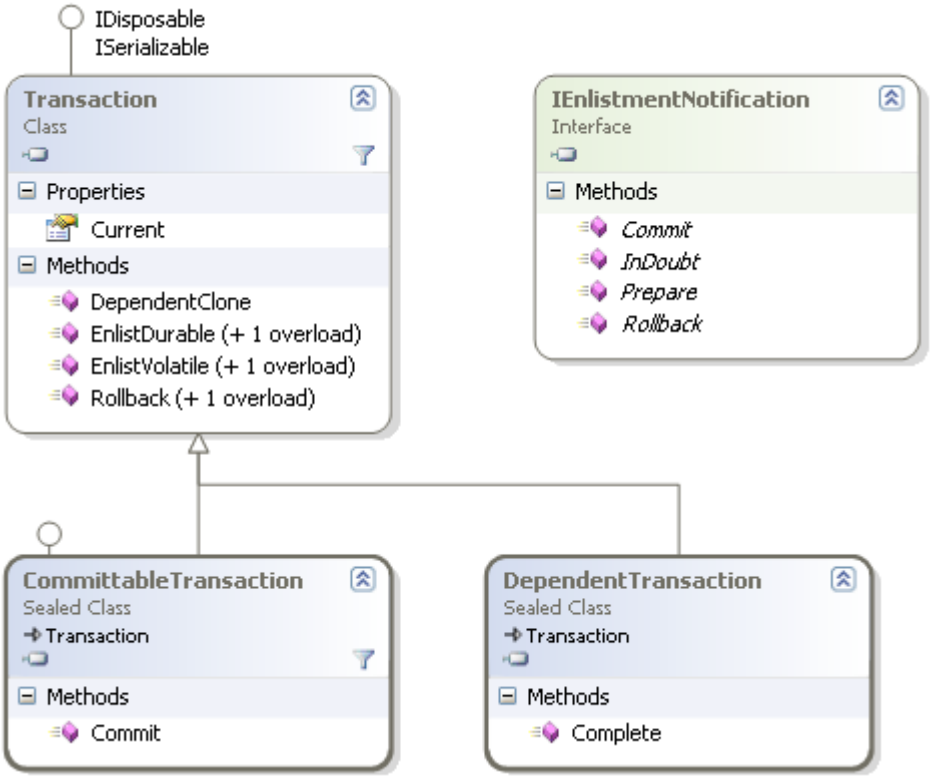

Figure 2.6: Transaction classes in .NET 2.0

3 Design and Implementation

3.1 Active Replication Per Partition Protocol

The Active Replication Per Partition Protocol (ARPPP) is after Primary-per-Partition Protocol (P4) and Adaptive Voting (AV) the next replication protocol following the ACBRM replication model in combination with active replication. The idea of applying ACBRM principles to Active Replication has been mentioned in Osrael's PhD thesis [Osr07], but besides an informal, unpublished description of the protocol by Osrael, no detailed published specification of the protocol exists. Section 2.2.3 introduces new phases added to the abstract replication model by ACBRM. In section 2.1.3 active replication is discussed in detail.

In the system architecture design for ACBRM described in [OFG+06] a membership service is supposed to keep track of node failures and system partitionings. Usually, a group membership service is combined with a group communication (GC) service. Therefore, ABCAST has been chosen to ensure server coordination in the SC phase, as a GC service is (usually) able to provide it.

However, an important goal is the support of transactions. In the context of .NET transactions the client can provide an ambient transaction when initiating a call to the system. In such case the system should behave as a transaction-aware component and join the transaction. This implies:

- distribution of the transaction to all nodes and
- transactional resource management (ensuring all the ACID properties as discussed in section 2.3) concluded with the 2PC protocol.

Compared to active replication using ABCAST from section 2.1.3, the 2PC is extra added. This is crucial, because the outcome of an operation is determined not only by the system, but also by external factors. Such factors can be the client's commit/abort decision, or the outcome of another external resource manager involved in the same transaction.

To summarize the ARPPP protocol phases, figure 3.1 illustrates them (for both normal and degraded mode) and a description of each phase follows:

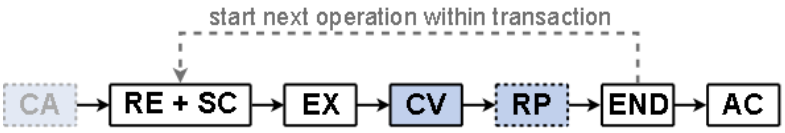

Figure 3.1: Phases of the ARPPP model.

CA (Configuration Adjustment) For the general ACBRM replication model, this phase is only required in degraded mode. However, ARPPP does not need to perform any adjustments in configuration during degradation, therefore this phase can be ommited completely.

RE+SC (Request + Server Coordination) This phase merges the sending of the operation request by client with server coordination. The client sends the request to all nodes using ABCAST, which implicitly ensures server coordination by receiving request messages on all nodes in the same order. Also, the client uses the ambient transaction and distributes it along with the request to all nodes. If no ambient transaction exists, the client starts a new one.

EX (Execution) The operation requested by the client is executed. Resources involved in the operation must be transactionally managed in order to achieve the ACID properties. The operations may initiate other (nested) sub-operations. These are handled following the same model, however a special synchronization is necessary as will be discussed later in section 3.6. The standard RNI problem (see section 2.1.3) must be handled too.

CV (Constraint Validation) This phase added by the ACBRM replication model validates consistency of constraints in the system immediately after executing an update operation. This phase is skipped if the executed operation was a read operation. As described in section 2.2.3, CV behaves differently in normal and degraded mode.

RP (Reconciliation Preparation) According to the general ACBRM replication model, this phase again is only required in degraded mode. Its task is to store information necessary for later reconciliation. For example, a log of the change performed during EX phase can be stored.

END (Response) After the operation has been processed, a response is

sent back to the client. Although, this ends the invocation, it does not finish the transaction. Because the client provided the transaction, it is also responsible for committing/aborting it. However, the client can decide to initiate another operation within the same transaction (see the loop back in figure 3.1).

AC (Agreement Coordination) After the client commits or aborts the transaction, the AC phase starts. Here, 2PC ensures the final agreement on all changes performed. At the end of 2PC all changes (and eventually the corresponding logs from RP phase) performed within the transaction are made durable. Also, all resources involved in the transaction are released.

3.2 System Architecture

The general architecture design [OFG+06] and its implantation into the .NET environment [OFS+06] have been used as a reference. Both proposals discuss the system components and main dependencies between them. The interceptor architecture of the Invocation Service (IS) as presented in [Hab06] and implemented in both P4 and AV has been used here as well (see section 3.2.1). Figure 3.2 illustrates the system design considering the interceptor architecture.

Some smaller components and some dependencies have been omitted for better readability. For example, a Lock Service is used by the Replica Manager and Transaction Manager, or the System Mode is visible to all components, which can adapt their behaviour according to it. Also, the Reconciliation Service uses other components to fulfill its tasks.

Design and implementation of the following components have been taken with less or more modifications from previous work done in frame of the DeDiSys project [Hab06], and the P4 and AV implementations:

- **Group Communication:** taken with almost no modifications.
- **Invocation Service:** taken with almost no modifications.
- **System Mode:** taken with almost no modifications.

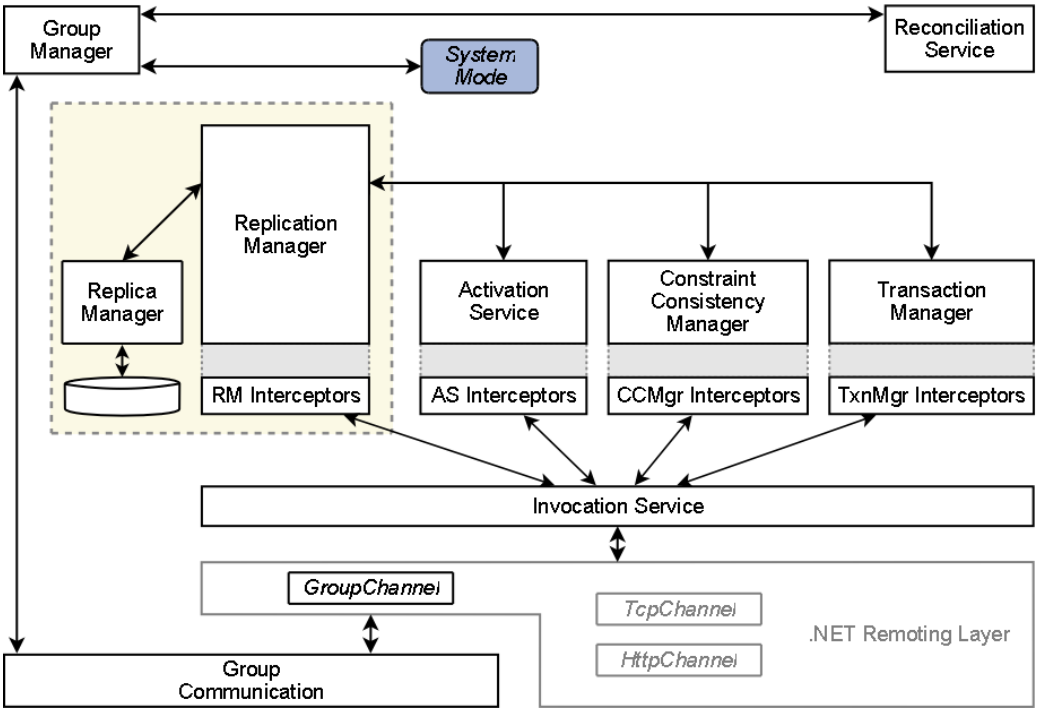

Figure 3.2: System architecture.

- **Activation Service:** taken with minor changes in implementation.
- **Reconciliation Service:** has been redesigned, however the logic of the algorithm for resolving replica conflicts has been kept.
- **Constraint Consistency Manager:** has been redesigned and partly reimplemented.

All other components have been redesigned and reimplemented completely or developed from scratch.

In the rest of this section an overview of each system component is given. The focus is on the tasks and dependencies to other components. Also, if necessary, any special design, tool, or technology decisions are presented.

3.2.1 Invocation Service

The Invocation Service (IS) provides means to intercept every method call placed on an object exposed by the system. Every system component is able to register its own *interceptors* with the invocation service. For one invocation, the client and server side, as well as before and after operation interception is distinguished. That means, four types of interceptors are available – *ClientBefore*, *ClientAfter*, *ServerBefore*, and *ServerAfter*. The sequence diagrams in figures 3.3 and 3.4 illustrate how the interceptor architecture works on the client and server side, respectively.

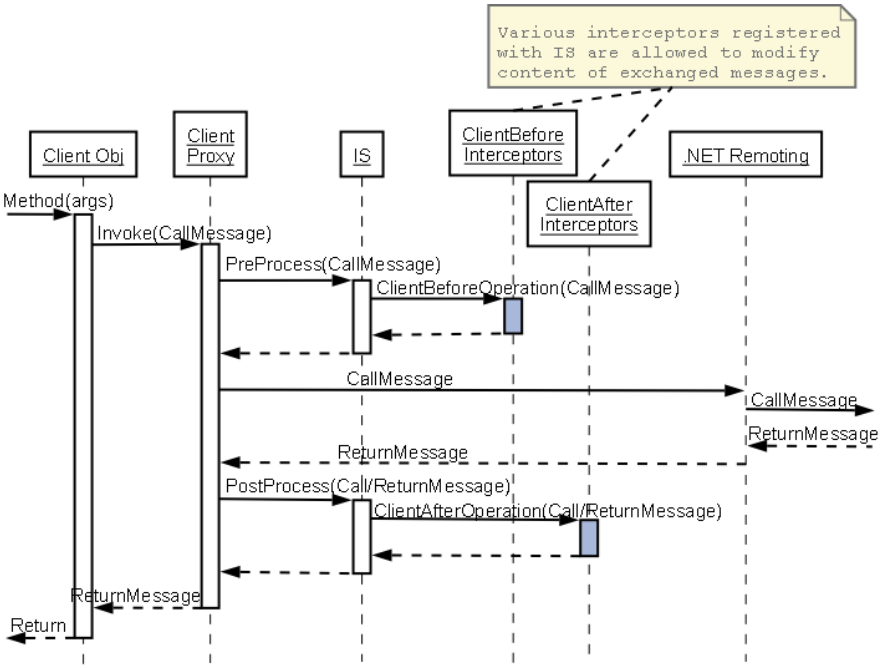

Figure 3.3: Interceptor architecture (client side).

This interceptor architecture allows every system component to append its part of handling to the overall call processing. .NET Remoting turns the client call into a *call message* object, which is supplied to all interceptors for custom processing. After the call is executed on the real object, the call result is represented by a *return message* object. This is therefore supplied only to the ServerAfter and ClientAfter interceptors.

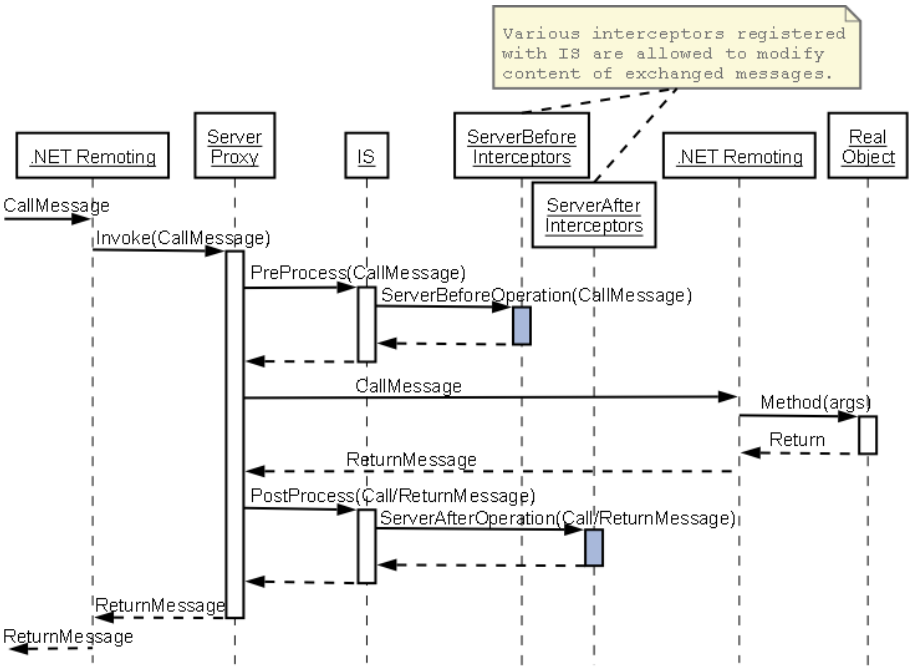

Figure 3.4: Interceptor architecture (server side).

Handling done by single components is described in appropriate subsections further below. From the view of IS, all handling done by the interceptors is optional. The only exception is, that any one of the ClientBefore interceptors must set the remote object's URI into the call message in order to address the call. As described later, this is done by the RM's ClientBefore interceptor (see section 3.2.6).

The remote invocation between two instances of IS is handled by the *.NET Remoting* layer. .NET Remoting offers different types of so called *channels* to transmit the call request/return message depending on the call data representation and the communication mechanism used (e.g. binary data/TCP or SOAP/HTTP). While these channels are appropriate for point-to-point communication, for active replication a means of group communication is advantageous. No such channel exists in the .NET Framework, however it is possible to provide a custom channel implementation.

For this purpose a custom *GroupChannel* enables two nodes to communicate

over a group communication service (see section 3.2.2). A client can address either the group as a whole, or any individual node. For the case of ordered message delivery, the GroupChannel assigns a sequence number to every call request received with a message. This enables higher level handlers to reconstruct the call request delivery order. A detailed description of group targeted invocations is discussed in section 3.5.

3.2.2 Group Communication

Group communication (GC) and a membership service is provided by the Spread Toolkit (http://www.spread.org) for the reasons explained in [OFS+06]. Among other features, Spread is able to provide both ABCAST and VSCAST as required by several protocol models described in section 2.1.

A group membership service in turn is necessary for handling node failures/rejoins and partition splits/reunifications as supposed by the ACBRM (see section 2.2.3). Spread's group membership service is further used by the Group Manager component described in section 3.2.3.

3.2.3 Group Manager

The Group Manager (GrMgr) handles all membership changes in the communication group. According to the state diagram presented in figure 2.2 (section 2.2.2), it triggers state transitions of the System Mode instance (see section 3.2.4). It is the only system component intended to do this. GrMgr is also responsible for starting the reconciliation process whenever necessary.

3.2.4 System Mode

System Mode is a publicly visible singleton (per node) instance providing information about the current mode the system is operating in. System components (if appropriate) adapt their behaviour according to the current state of the System Mode.

3.2.5 Activation Service

One task of the Activation Service (AS) is to exchange client provided real objects for proxy objects exposed by the system. This enables every call to be intercepted by the IS as described in section 3.2.1.

The second task of the AS is to activate the instance targeted by an invocation. In this case, the AS obtains the instance from the RM. (Other features like passivation of unused instances is possible, however not intended for the prototype.) The activation is accomplished by registering a **ServerBefore interceptor** with IS, which activates the target object prior to the call.

3.2.6 Replication Manager

The core component responsible for replica management and replication is the Replication Manager (RM). It manages the local replicas, keeps track of their versions, and implements the replication protocol.

The RM uses a separate component called *Replica Manager* to manage replica objects[3]. Although *isolation* (from the ACID properties) should be ensured by the replication protocol during server coordination, it is (redundantly) guaranteed by the Replica Manager as well. For every operation, the Replica Manager requires a suitable lock to be owned by the executing transaction. This does not harm performance and provides a general approach with high robustness guarantees.

Resources involved in an operation are enlisted with the executing transaction as volatile resources (see section 2.3.3). Durable resource management is much more complex and not necessary for the objectives of this thesis. Each resource participates in the 2PC protocol ensuring *atomicity* and *durability* of the transactional resource management as well.

[3]The *Replica Manager* stores all replica objects on one node, ensures their backup in a persistent storage, keeps track of their versions, ensures concurrency control by requesting replica locks, and provides transactional management of replicas. On the other hand, the *Replication Manager* implements the replication protocol itself using the features of the *Replica Manager*.

The Replica Manager stores the replica objects in runtime memory, but keeps synchronized copies in a database, too. This enables to load all durably persisted resources even after a node crash.

Another task of the Replica Manager is to log the history of replica versions during degraded mode. This can be done in several ways as described in [OFG07]. Logging full history of changes allows to use any reconciliation strategy. In order to provide as much configuration options for the prototype as possible, this approach has been chosen.

Finally, the most important task of the RM is the management of the replication process itself. It is realized by scheduling the operations to be executed. For this purpose, the RM registers all four types of interceptors. A brief overview of each interceptor's task follows, however, a detailed description of the scheduling process is described in section 3.6.

ClientBefore interceptor is responsible for selecting the target of the remote invocation. In case of the ARPPP, this is the whole communication group. For nested invocations, this interceptor also performs some synchronization with the parent call, needed for proper scheduling, as will be discussed later.

ServerBefore interceptor is the most important interceptor. It schedules the execution of all operations according to the operation receipt order. It does this by suspending each operation's thread until it is allowed to execute. The scheduling must happen in a deterministic way, in order to achieve the same operation execution for one replica order on all nodes.

ServerAfter interceptor does some synchronization needed for proper scheduling. This will be discussed later in more detail.

ClientAfter interceptor performs for nested operations some synchronization with the parent operation needed for proper scheduling. This will also be discussed later in section 3.6.3 in more detail.

3.2.7 Constraint Consistency Manager

The Constraint Consistency Manager (CCMgr) holds all constraints defined in the system and ensures constraint consistency according to the rules described in section 2.2.3.

To validate constraint consistency in the system, the CCMgr installs a **ServerAfter interceptor**, which approves changes made by a write operation immediately after execution. As an optimization, the CCMgr also installs a **ClientBefore interceptor**, which aborts any *critical operation* during degraded mode.[4]

3.2.8 Transaction Manager

The Transaction Manager (TxnMgr) is designed to provide transactional support for the replication process on the local node. It keeps track of every distributed transaction since it has been first seen on the node, until the transaction finishes.

The TxnMgr associates a *TxnContext* with every transaction seen on the particular node. The TxnContext stores transaction specific data like, for example, locks of resources involved in the transaction.

TxnMgr also registers all four types of interceptors with the IS. Their tasks are as follows:

ClientBefore interceptor is the very first client-side interceptor. It installs an ambient transaction, which can be then used by all other client-side interceptors on this thread.

ServerBefore interceptor is the very first server-side interceptor. It uses the transaction distributed by the client with the operation request and installs an ambient transaction. This can be subsequently used by all other server-side interceptors on the thread.

[4]The operation criticality can be checked on the client side because in the prototype every node is a client and a server at the same time. If this was not so, the check must happen within a ServerBefore interceptor.

ServerAfter interceptor as the very last server-side interceptor finishes the ambient transaction installed by the ServerBefore interceptor.

ClientAfter interceptor is the very last client-side interceptor that finishes the ambient transaction installed by the ClientBefore interceptor.

Transaction processing is discussed more closely in section 3.4.

3.2.9 Reconciliation Service

The Reconciliation Service (RS) is the component responsible for replica consistency reestablishment after two partitions rejoin. If the partition reunification results in a healthy system, the RS is also responsible for constraint consistency reestablishment as discussed in section 2.2.3. Reconciliation is triggered by the Group Manager when a failed node recovers or partitions rejoin.

As the reconciliation process is not in the focus of this thesis, only a very simple reconciliation algorithm (based on the *more updates win* strategy) has been taken from the AV implementation described in [Chl07].

The RS tightly cooperates with the RM to gather all data and information necessary for the reconciliation. Because the RS needs to have exclusive access to all resources, it has special priviledges to "turn off" the TxnMgr.

3.2.10 Interceptors overview

After all main system components and their registered interceptors have been described, a comprehensive overview of all interceptors is presented. Figure 3.5 illustrates which *client-side* interceptors are called by the IS and in which order. Figure 3.6 in turn illustrates the *server-side* interceptors and their order as they are called by the IS.

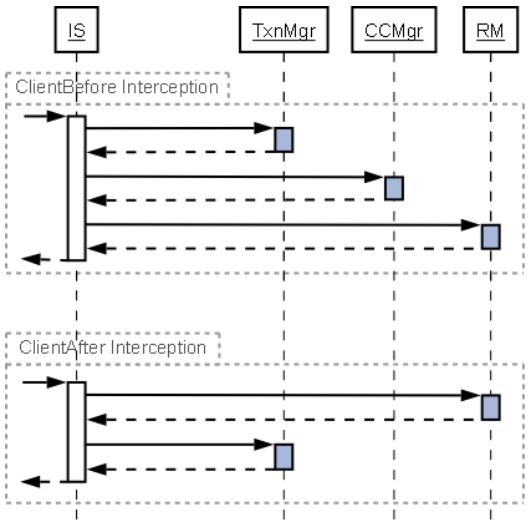

Figure 3.5: Client-side interceptors overview.

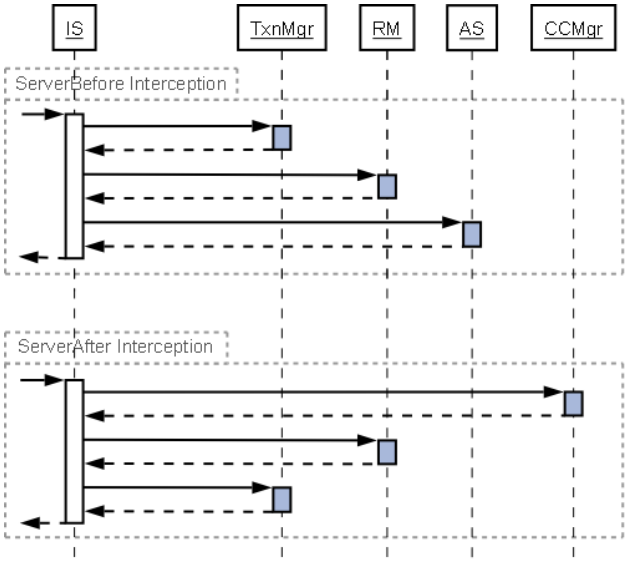

Figure 3.6: Server-side interceptors overview.

3.3 Replica Objects

The concept of a replica object has been basically taken from previous solutions (P4 and AV) and is represented by the class `ReplicableObject` illustrated in figure 3.7.

Figure 3.7: The `ReplicableObject` class.

`ReplicableObject` is an abstract class supposed to be extended by user defined classes, that should have the "replication ability". For this purpose, every replica object needs to have a unique identifier (`ObjectId`) and a version identifier (`Version`). Both these members are used exclusively by the system and are not visible to the outside world.

The Activation Service provides means for exchanging a client supplied `ReplicableObject` for a client proxy substitute. In this case, a client and a server proxy for this object are created and a new unique `ObjectId` is assigned to both proxies, as well as to the replica object. Then, the real instance is passed to the Replication Manager (RM), where it is further managed (see section 3.2.6). The client proxy is passed to the client to be used for invocations and the server proxy is published by the .NET Remoting infrastructure to accept remote invocations.

The `ReadMethodAttribute` attribute is used by user extensions of `ReplicableObject` to mark methods, that do not perform any changes of the replica object. Such operations are then treated by the system in a

more efficient way, according to the ACBRM replication model specification (see section 2.2.3).

In contrast to the previous implementations, one of the goals here is to enable nested invocations. For this purpose, inter-replica references must be supported. To achieve the desired behaviour, a replica A referencing another replica B must store B's proxy object instead of the real instance of B. This ensures a nested call to B to be intercepted by the system and handled appropriately.

To store a replica proxy instead of its real instance is a matter of object construction. However, replica objects are serialized frequently, so the serialization process must not destroy the proxy reference. Because custom proxy implementations are used in the system, custom serialization is necessary, too. For this purpose, every custom `ReplicableObject` implementation must implement the `ISerializable` interface. For the serialization and deserialization of replica references, `ReplicableObject` provides the `GetReplicableObjectData` and `ReplicableObjectFromData` methods, respectively.

In fact, when serializing a replica proxy, only the corresponding `ObjectId` is recorded. During deserialization then, the proxy for the `ObjectId` must be found and assigned.

3.4 Transaction Support

3.4.1 Component Details

As described in section 3.2.8, transaction support during invocations is implemented by using interceptors. For this reason, it is not suitable to use the `TransactionScope` class for transaction management. Instead, the more complex technique of using plain transaction objects must be used (see section 2.3.3). The Transaction Manager (TxnMgr), however, provides a simplified interface for managing transactions, as shown in the class diagram in figure 3.8.

StartTxn is a transaction-aware method that creates a new `CommittableTransaction` object or, if an ambient transaction is present, clones a `DependentTransaction` from it. Optionally, the caller may pass in

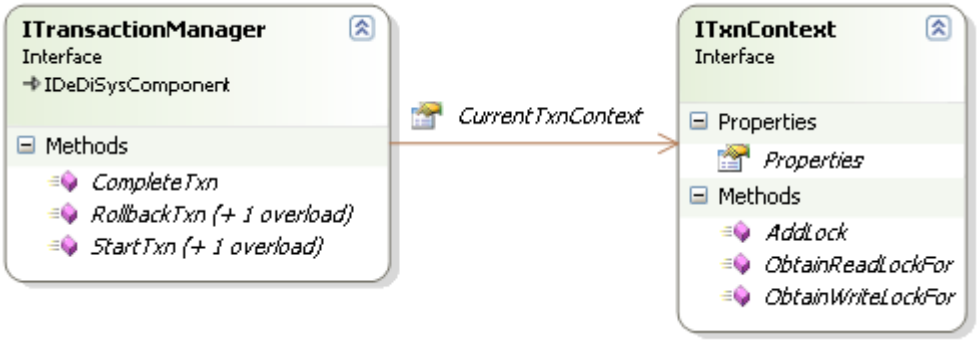

Figure 3.8: The `ITransactionManager` and `ITxnContext` interfaces.

a `Transaction` object from which the dependent should be cloned. This method also sets the new `Transaction` object as the ambient transaction and pushes the previous one onto a stack.

Calling the *Rollback* method rolls back the ambient transaction. An `Exception` object may be optionally passed in to indicate the cause for aborting the transaction. At the end, this method resets the previous ambient transaction by popping it from the stack.

After all work within the ambient transaction has been successfully done, the *Complete* method must be called. If the ambient transaction is a `CommitableTransaction`, it is committed. Otherwise, if it is a `DependentTransaction`, it is completed. Afterwards, this method pops the previous ambient transaction from the stack and resets it.

3.4.2 Transaction Context

The `Transaction` class provides basic information about the transaction it represents like a local ID, a global ID, or a transaction status. To store other transaction specific information, the *Transaction Context (TxnContext)* has been designed. It is specific for one (logical) transaction and one node. I.e. the TxnContext is also node specific. The `ITransactionManager`'s field `CurrentTxnContext` provides the TxnContext associated with the ambient transaction (see figure 3.8).

The TxnMgr associates a new TxnContext with every transaction, when it is first seen. At creation time, TxnContext enlists with the corresponding transaction and so becomes involved in the 2PC. Although it is part of the 2PC protocol, TxnContext never votes for aborting the transaction. When the transaction finishes, the TxnContext simply disposes itself during the commit, or as the case may be, the rollback phase.

The main task of TxnContext is to store resource locks owned by a transaction. These locks are then requested by the Replica Manager to control concurrent access to replica objects. More on the locking system is discussed in section 3.4.3. As declared by its interface (see figure 3.8), TxnContext provides methods to add and obtain locks. All locks held by a TxnContext are released at dispose time, when the transaction finishes.

Also, TxnContext provides a general purpose field `Properties`, where arbitrary data can be stored.

3.4.3 Lock Service

The only resource type in the system are replica objects. Replicas are locked on each node independently by acquiring locks at a node-central component called the *Lock Service*. As figure 3.9 shows, two types of locks can be acquired – *read locks* (`IReadLock`) and *write locks* (`IWriteLock`).

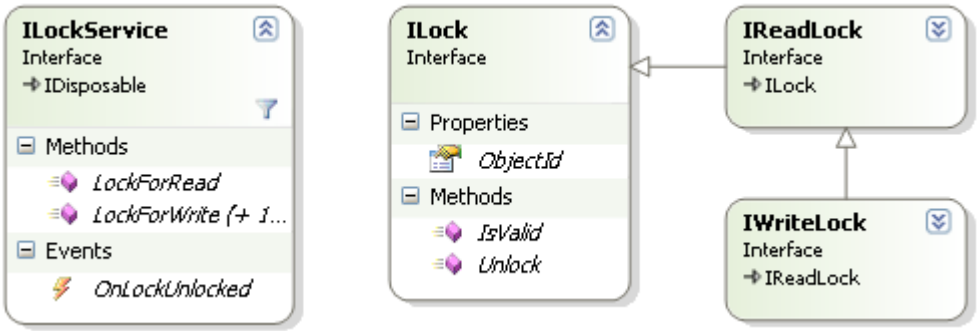

Figure 3.9: Lock Service interfaces.

The Lock Service must ensure the usual rules for lock issuing. At the same time, for one replica (on one node) the following must be satisfied:

- Either no lock is issued, or
- any number of read locks, but no write lock is issued, or
- one write lock, but no read lock is issued.

The Lock Service interface (see figure 3.9) provides methods to request both types of locks. Additionally, when requesting a write lock, the caller can pass in the read lock it already owns. In this case, the Lock Service tries to *advance* the given read lock to a write lock. All methods are nonblocking and return `null` in case the lock requested cannot be issued.

A lock is mapped to a replica by holding the replica's `ObjectId`. The base interface of a lock (`ILock`) enables its holder to release the lock as well as to check, whether the lock is still valid. The former method is used by the TxnContext at dispose time, while the latter is used by Replica Manager to check the validity of a lock provided for an operation.

When a lock is unlocked, the Lock Service sends the `OnLockUnlocked` event to all registered delegates containg information about which replica has been unlocked and whether it is now possible to acquire a new lock.

3.4.4 Resource Management

As already discussed, the Replica Manager controls the access to all replica objects on a certain node. On one hand, this incorporates *concurrency control* ensured by using a standard locking mechanism with read and write locks. On the other hand, *transactional resource management* is required as well.

Besides other functionality, Replica Manager offers the `GetReplica` method to obtain a read copy of a replica object and the `UpdateReplica` method to install a new version of a replica passed in as a parameter. To secure concurrency control, the former method provides the replica's read copy only if the ambient transaction owns the appropriate read lock. Analogously, the latter method installs the provided replica object only if the ambient transaction owns the appropriate write lock. How the locks are obtained is described in section 3.4.2.

To ensure transactional ACID properties, the Replica Manager's `UpdateReplica` method installs a new replica version in a transactional manner. When the method is called, the new replica version is wrapped by the `UncommittedReplica` object, which is then enlisted with the ambient transaction and stored in TxnContext. In this way, when the transaction finishes, the wrapper object will participate in the 2PC, which is necessary for providing *atomicity* and *durability*. Apart from that, storing the temporal changes in TxnContext ensures *isolation*, because every transaction has access to its own (and no other) TxnContext and thus does not see changes made by other concurrent transactions.

For resources, that only need to be notified about the transaction end, but do not require to vote on its outcome (similar to TxnContext), the abstract class `DisposableTxnResource` has been designed. Its subclasses automatically enlist with the ambient transaction at construction time and implement the `Dispose` method, that is called when the transaction finishes. The use of this class will be presented later in section 3.6.3.

3.4.5 Transactions in the ARPPP

After the concepts, components, and mechanisms supporting transaction processing have been presented, this section describes their interplay within the replication using the ARPPP.

When the client calls a method on a replicated object an ambient transaction may or may not be running. The very first interceptor called is the TxnMgr's ClientBefore interceptor. This calls the TxnMgr's `StartTxn` method, which installs a newly created `Transaction` object as the ambient transaction (see section 3.4.1). Also, the TxnContext for this transaction on this node is created. All the subsequent client-side interceptors have the access to the ambient transaction as well as to its TxnContext.

Except that, the TxnMgr's ClientBefore interceptor copies the new ambient transaction into the call message. In this way, the `Transaction` object is broadcasted to all server nodes in the call message. Each server node starts processing the call message with the TxnMgr's ServerBefore interceptor as the very first server-side interceptor. Using the broadcasted `Transaction` object, this interceptor starts a new dependent transaction and sets it as the

ambient transaction. The TxnContext is created too. From now on all server-side interceptors can access the ambient transaction and the corresponding TxnContext.

Evenmore, the same thread performs the method invocation on the real object. Therefore, if the replica object contains transaction-aware methods, these automatically join the ambient transaction and become a part of it. This is especially useful for nested invocations. If the replica object places a call to another replica object, this nested call happens already in the context of the ambient transaction started by the system.

The transaction mechanism described for nested invocations clones always the same transaction and thus ensures the execution within the same logical transaction. This causes a call and its nested call to have a common TxnContext, which is essential for proper transactional resource management.

TxnMgr's ServerAfter and ClientAfter interceptors finish the ambient transaction at the end of server-side and client-side invocation processing, respectively. If an ambient transaction has been started by the client prior to the replica method invocation, the client is responsible for finishing the transaction. The client can also decide to abort the transaction (although the invocation has been successful), or even start another invocation within the same transaction.

3.5 Group Targeted Communication

Another important part of the system is the support for invocations targeting the whole communication group, instead of a single node. This is needed for simple call replication within the active replication scheme. As already discussed in section 3.2.2, the Spread toolkit provides a group communication service with the message broadcast ability. How a custom channel implementation can be integrated into .NET Remoting is suggested in section 3.2.1. Here, the concrete design and implementation of such a channel is described.

3.5.1 Group Channel Requirements

Using the underlying group communication service of Spread, the custom group channel must:

- be able to broadcast a call message to every node in the communication group,
- accept call messages sent to the channel, process them, and send return messages back,
- handle the Redundant Nested Invocation (RNI) problem for duplicated call and return messages,
- allow the caller to specify for how many replies should be waited before making the return message available (for the first reply only, or for all replies)
- assign a sequence number to every call message received.

To be compatible with the .NET Remoting infrastructure, the group channel must further implement the following three interfaces:

`IChannel:` Methods exposed by this interface are used by .NET Remoting for general channel management. For example, for obtaining the channel's name and priority, or to check whether an invocation URL is accepted by a channel or not.

`IChannelSender:` This interface exposes only one method (`CreateMessageSink`) to be used by a client wanting to send a call message using this channel. The method must return an `IClientChannelSink` implementation, that can be further used to send call messages. An URL of the invocation target is passed in to adjust the sink behaviour. The format of the URL for the custom channel is described in section 3.5.2.

`IChannelReceiver:` Methods declared here are needed for the channel's server behaviour management. For example, the `StartListening` and `StopListening` methods enable or disable the channel's ability to receive and process messages.

3.5.2 URL for Invocations

In the .NET Remoting infrastructure every channel supports a specific URL format to address the invocations. The URL format for the group channel is the following:

gcp://<group_id>/<target>/<object_uri>

The string *gcp* at the very beginning is an identifier of the URL format and stands for *g*roup *c*ommunication *p*rotocol. The following *group_id* denotes a concrete group as the underlying group communication service can manage any number of different groups. The *target* of the invocation (within the specified group) can be defined in two ways. Using the string *all* will broadcast the invocation to all members of the specified communication group. Alternatively, one node can be addressed by using its IP endpoint identifier in the form <*ip_adr*>:<*port_nr*>. The rest of the URL is the application specific object identifier (*object_uri*) denoting the object to be used for the invocation.

3.5.3 Interface to Group Communication Service

Access to a group communication service is basically provided over the two interface definitions presented in figure 3.10. Concrete implementations access some real group communication service like Spread.

`ICommunicationGroup` is an abstraction of a communication group. Except of providing group specific information, it is able to establish connections to the group. Such a connection is represented by implementation of the `IGroupConnection` interface. Also, `ICommunicationGroup` is supposed to raise events when group members leave or join the group. This is exploited by the Group Manager as described in section 3.2.3.

The `IGroupConnection` object then provides means to multicast messages to the group members. If a connection receives a message, it raises the `OnMessageReceived` event.

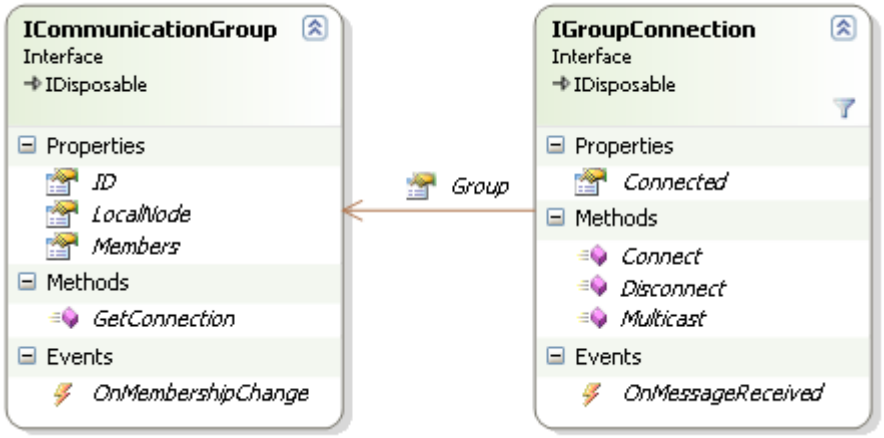

Figure 3.10: Interface to group communication service.

3.5.4 Group Channel Description

Figure 3.11 illustrates the main classes involved in the mechanism providing group targeted communication.

The central class here is the `GroupChannel` class, which implements the three channel interfaces discussed in section 3.5.1. When creating an instance of this class, an `IGroupConnection` instance must be passed in, to associate the `GroupChannel` with a communication group.

A remote call is placed by exchanging a pair of `CallRequest` and `CallResponse` objects between the client and server `GroupChannels`. The `CallRequest` transmits the call message to the server, while the `CallResponse` transmits the return message back to the client. Every call is identified by a `CallID`. With every call a `CallControl` object is associated to store call specific information needed for proper call handling.

In the rest of this section, the single processes of sending and receiving call and return messages are described with focus on the requirements discussed in section 3.5.1.

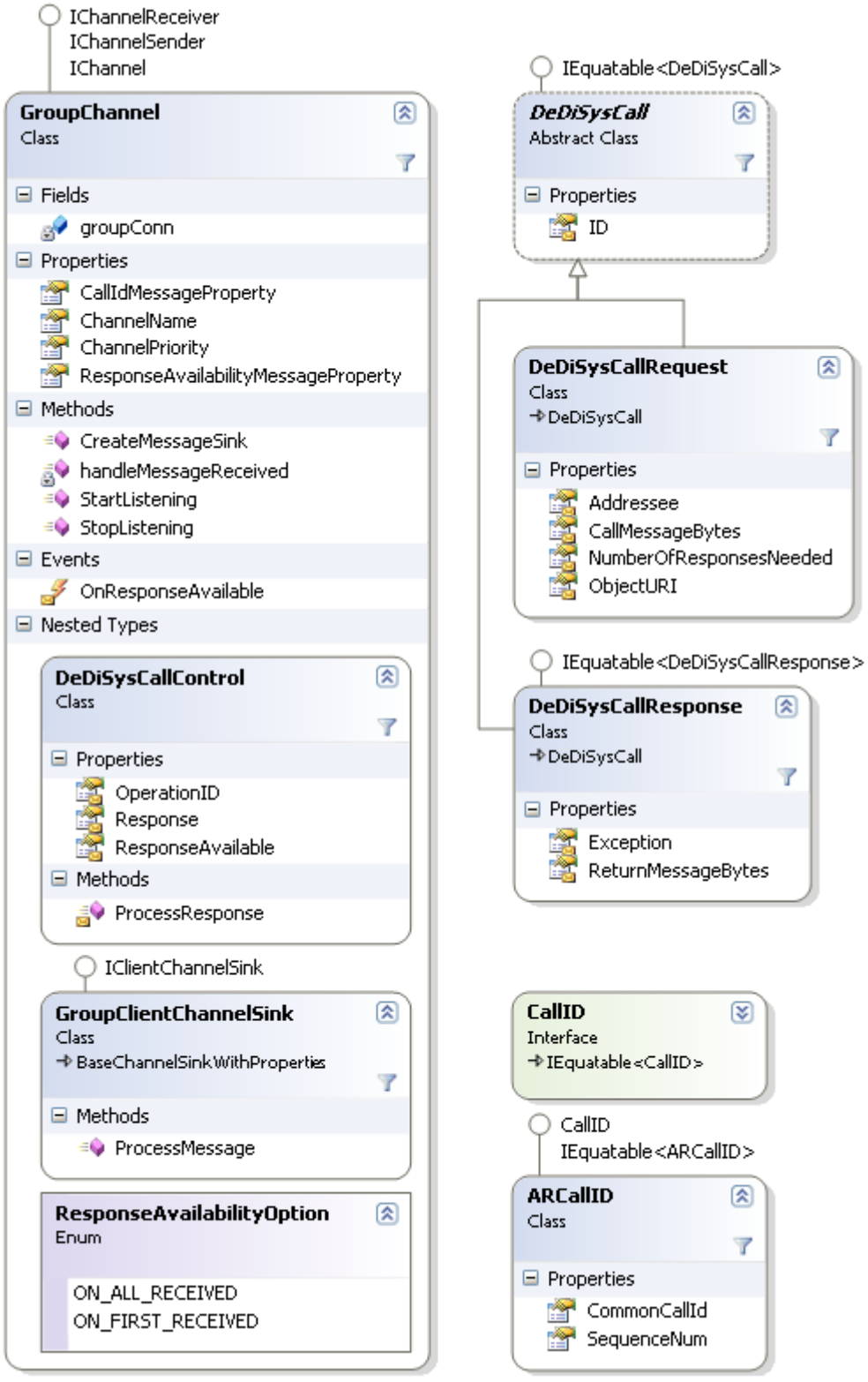

Figure 3.11: Classes providing group targeted communication.

Registering the Group Channel

To integrate the described group channel implementation into the .NET Remoting infrastructure, the channel must be registered by using the `ChannelServices.RegisterChannel` method.

Sending the Call Message

When the client proxy intercepts a call, the Replication Manager's ClientBefore interceptor sets the URL of the invocation target. It also provides a custom `CallID` and the desired `ResponseAvailabilityOption`. Using the URL, the client proxy finds the corresponding group channel. By invoking its `CreateMessageSink` method, the client obtains an `IClientChannelSink`. This, in fact, is an instance of the nested class `GroupClientChannelSink` (see figure 3.11).

The custom `IClientChannelSink` implementation provides only the `ProcessMessage` methods for synchronous invocations. For the sake of simplicity, other methods for asynchronous invocation are not implemented.

When the client proxy calls the `ProcessMessage` method, it tries firstly to extract the custom `CallID` from the provided call message. If none is found a new one is generated. Then it checks, whether a `CallControl` for the `CallID` exists. If yes, this means the call message has been already received from some other node and thus a redundant sending is suppressed (this handles the RNI problem for call messages). Otherwise, the call message (together with the `CallID`, addressee, URL, and number of responses needed[5]) is packed into a `CallRequest` object and multicasted to the group.

At this stage, the `CallControl` object is *not* created. This is an exclusive task of the call message receiving process. For this purpose, it is necessary to always multicast the message to the own node as well.

On the client side, the process continues with receiving the return message, which is described below.

[5]number of responses needed to make the response available to the client, according to the `ResponseAvailabilityOption`

Receiving the Call Message

At the server side, after receiving the `CallRequest`, the channel checks whether the message is addressed to it. If so and if it has not been already received, a sequence number is assigned to this request and a `CallControl` object is created. Then, a handler is started to process the request. If a `CallControl` object already exists, this means the call request has been already received and executed and thus it must not be handled once again. In this case, the handler skips to the sending of a return message.

Otherwise, the handler first unpacks the call message and sets the sequence number assigned by the channel into the message. Then it finds the server proxy of the object denoted in the URL. Afterwards, by using .NET Remoting, the call message is used to place a call on the server proxy, which invokes all interceptors, as well as the target object.

Sending the Return Message

After the call has been executed, the handler checks the call's `CallControl` object, whether it is still needed to send the response. If the requested number of responses has been already received (from other server nodes), sending the response would be redundant and thus is suppressed (this handles the RNI problem for return messages). Otherwise, the handler packs the return message into a `CallResponse` object and sends it back to the group.

Receiving the Return Message

The `GroupClientChannelSink` on the client side after sending (or suppressing the sending of) the call message waits until the `CallControl` object makes the response available. After that, it forwards the return message back to the caller of the `ProcessMessage` method.

The `CallControl` object is notified about received `CallResponse`s by the `GroupChannel`'s `handleMessageReceived` method, which is a registered delegate for the `IGroupConnection.OnMessageReceived` event. According to the `ResponseAvailabilityOption` of the request, the response is made available after enough `CallResponse`s have been received.

3.6 Replication and Call Processing

Now, after all the supporting mechanisms have been presented, this section describes how the replication process works. The central component here is the Replication Manager (RM) with its interceptors, which are briefly discribed in section 3.2.6.

3.6.1 Replication Process Requirements

For the replication process, the most important tasks of the RM are:

- to replicate client calls to all server nodes,
- to schedule the call execution in a deterministic way, so that the operation order for one replica is identical on all nodes,
- to allow execution parallelism to the largest extent possible, and
- to optimize the processing of read calls.

Call replication is easily achieved by the RM's ClientBefore interceptor by setting a target URL that uses the group channel and addresses all nodes in the current communication group. It is also important, that the underlying group communication service is set to provide atomic broadcast (ABCAST) guarantees for message delivery.

The deterministic call scheduling on one replica (with respect to high parallelism) is not so easy to achieve. ABCAST used for the call replication ensures the same delivery order of call requests on all nodes. However, by using .NET Remoting, a separate thread is started to handle a call request. Assuming this, the operating system provides no guarantees regarding the scheduling of threads running concurrently. Therefore, a system of *call guards* has been designed, which schedules the thread execution, while making use of the sequence numbers assigned to every call message by the group channel (see section 3.5).

Under certain circumstances, optimization of read call processing is possible and desired. It will be explained in section 3.6.5.

3.6.2 Concept of a Common Call

Before the call guard system will be described, the concept of the so called *common call* must be introduced. A common call is the union of all calls belonging to one logical transaction. In fact, there is a one-to-one mapping between a transaction and a common call.

Every common call has a unique identifier. Sequence numbers are assigned to single calls in the order they are issued, so a single call is identified by its common call's ID and a sequence number. This identifier construction mechanism has the important characteristic, that if one replicated call executing on several nodes initiates a nested call, the same ID is assigned to all nested calls produced without any synchronization. The call ID is represented by the class `ARCallID` as shown in figure 3.12. `ARCallID` implements the `CallID` interface introduced in section 3.5.4 and thus can be used with the group channel.

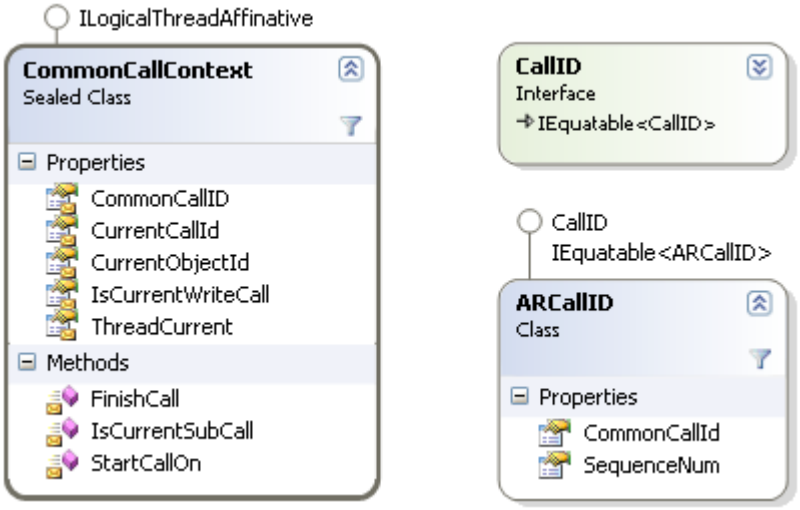

Figure 3.12: The common call context and call identifier classes.

To provide a common call context, an instance of `CommonCallContext` (RM's nested class, see figure 3.12) is sent along with all calls belonging to one common call. Similar to the Transaction Manager, the RM's interceptors install the current call's `CommonCallContext` into the threadstatic field `CommonCallContext.ThreadCurrent`. In this way, the `CommonCallContext` of the current call is accessible to everyone working on the thread.

The `CommonCallContext` provides information about the current call, however, it also stores data about all its parent calls. The data stored about every single call include:

- call ID,
- `ObjectId` of the targeted replica,
- a flag whether it is a write or a read call, and
- a flag whether it is a nested call or not.

Additionally, `CommonCallContext` provides methods to start a new (eventually nested) call on a specified replica (`StartCallOn`) and to finish the current call (`FinishCall`). Starting a new call is done by the RM's ClientBefore interceptor. It also automatically generates a new call ID and pushes the eventual parent call onto a stack. In turn, the RM's ClientAfter interceptor finishes the call. If it was a nested call, this resets the parent call to be the new current again.

3.6.3 Call Guard System

The designed call guard system is a mechanism to ensure a proper call execution order on single replicas, i.e. on all nodes the same order of operations executed on one replica. This is one of the basic requirements of active replication to achieve deterministic processing with respect to concurrent calls.

Additionally, (nested) subcalls must be handled as well. To allow the execution of a subcall even on a single replica, the concept of *suspending* a (parent) call has been considered as well.

Figure 3.13 shows an example when a call to replica A (*call1*) starts a subcall to replica B (*call2*), which in turn starts another subcall back to replica A (*call3*). On replica A, this causes the *call3* being executed "within" the *call1*. With active replication, the *call3* request could arrive from another node even before *call2* has been initiated locally (*ALT-call3*). Also, the response to *call2* (*ALT-call2/ret*) could arrive before *call3* finishes locally. Both situations would cause indeterministic concurrency behaviour.

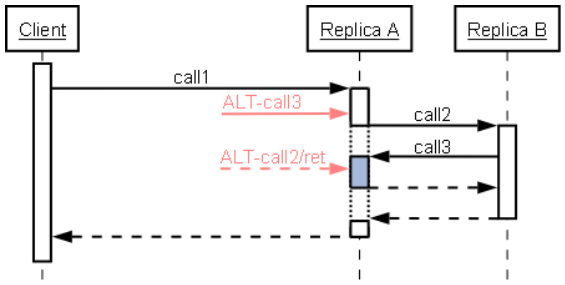

Figure 3.13: Necessity of suspending the parent call.

To avoid such problems, the underlying communication service could provide *causal atomic broadcast (CABCAST)* for message delivery. In this case, a subcall message would not be delivered on a node until the corresponding parrent call message has been delivered. The Spread group communication toolkit is supposed to provide CABCAST, however, to the knowledge of the author, there exists no documentation on how to use this feature.

Therefore, the call guard system has been designed to synchronize the calls in such scenarios as well. Nevertheless, no significant overhead should appear in case the communication service provides CABCAST[6].

A Call Guard Component

A call guard is a component exposing methods, that are called during the processing of *every* call. By calling these methods, the call guard system is able to discriminate different phases of a call's lifecycle. Some of these methods only notice the call's phase changes, while others may also block the thread's execution until some criteria are fulfilled.

Figure 3.14 shows the classes involved in the call processing and the call guard system. All call guard components implement the `ICallGuard` interface[7]. For every single call, on one `ICallGuard` component, the methods must be invoked in the following order:

[6]In fact, the custom mechanism combined only with ABCAST is expected to perform better, because the synchronization takes place further in the invocation process.

[7]For the sake of lucidity, in figure 3.14 the interface methods are ommited in all concrete classes implementing `ICallGuard`.

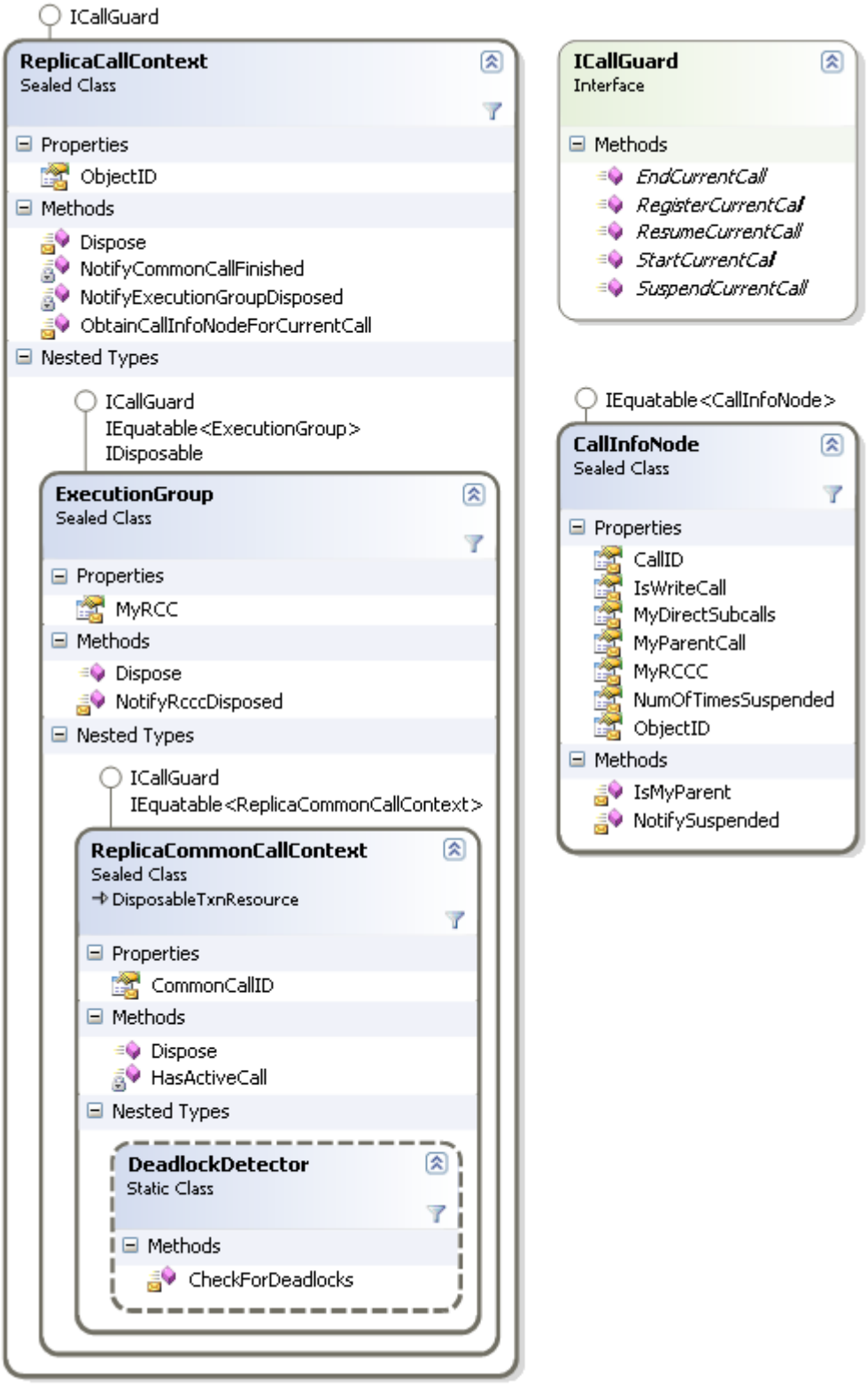

Figure 3.14: Call processing classes.

1. `RegisterCurrentCall` (mandatory)
   Registers the current call on the component. If the registration order for two calls is equal on different nodes, then all the other methods behave in the same way on these nodes regarding the two calls.

2. `StartCurrentCall` (mandatory, may block the calling thread)
   By calling this method, a previously registered call "asks for a permission" to start its execution. Different `ICallGuard` components may require different criteria to be fulfilled, in order to grant such permission. Until the permission is granted, the calling thread is suspended.

3. `SuspendCurrentCall` (optional)
   Optionally, the component might need to be informed, that a call has been suspended (e.g. when a call initiates a subcall).

4. `ResumeCurrentCall` (optional, may block the calling thread)
   If a call has been suspended, it needs to be resumed before continuing in work. Similarly to `StartCurrentCall`, this method may block the calling thread, until some internal criteria are fulfilled (e.g. the subcall ends).

5. `EndCurrentCall` (mandatory)
   Calling this method notifies the component, that the current call has finished all its work.

The `SuspendCurrentCall` and `ResumeCurrentCall` methods can be optionally invoked multiple times (however, both and in the specified order). For example, in case a call initiates several subcalls during its execution.

Interaction with Replication Manager

In the designed call guard system, there are three types of components implementing the `ICallGuard` interface (see figure 3.14). The RM's interceptors use only objects of type `ReplicaCallContext` to synchronize concurrent thread execution. Objects of other `ICallGuard` implementations are used internally by `ReplicaCallContext`. In the following, the tasks of all RM's interceptors with respect to the call scheduling and synchronization are described:

ClientBefore Before starting a new call, this interceptor suspends the parent call (if there is any) by calling the `SuspendCurrentCall` method on the appropriate `ReplicaCallContext` object.

ServerBefore This interceptor registers and starts every call. It ensures, that for *any* two concurrent calls, the `RegisterCurrentCall` method (of the appropriate `ReplicaCallContext`) is invoked in the order they have been received. This is done by making use of the call's sequence number assigned by the group channel and stored in the call message (see section 3.5). After a call has been registered, the `StartCurrentCall` method is called on the appropriate `ReplicaCallContext` to notify that the call starts. The method will block execution until the call is allowed to start.

ServerAfter After the call has been executed on the server, this interceptor finds the call's `ReplicaCallContext` and invokes its `EndCurrentCall` method to indicate the call has been executed.

ClientAfter Only in the case the current call is a subcall, the parent call must "ask for a permission" to resume execution. Therefore, `ResumeCurrentCall` method of the parent call's `ReplicaCallContext` is invoked. This method will block execution until the parent call is allowed to resume execution.

Detailed Description

After the interaction between the RM's interceptors and the call guard system has been discussed, the call guard system can be described in detail. It consists of three types of `ICallGuard` components – *Replica Call Context (RCC)*, *Execution Group (EG)*, and *Replica Common Call Context (RCCC)*. These are implemented by corresponding classes illustrated in figure 3.14.

When a call request arrives, it must be associated with exactly one RCC, one EG, and one RCCC. A strict hierarchy exists between these components. When a RCC's call guard method does its work for a certain call, it invokes the corresponding method on the call's EG. In turn, when the call's EG does its work, it invokes the same method on the call's RCCC. In the following, each component is described, focusing on its internal criteria of when a call is allowed to execute:

Replica Call Context (RCC) For every replica installed on a certain node, a RCC is created. The RCC is responsible for the first-level call ordering on the corresponding replica. It does this by associating common calls with EGs (described later). RCC also maintains a queue of EGs. When a call request arrives, it is firstly checked whether its common call has already an associated EG within this RCC. If not, the RCC tries to associate (register) it with the last EG in the queue. If this fails, a new EG is created, associated with the current call's common call, and inserted into the queue. The RCC allows a call to be executed only if its EG is the first in the queue.

Execution Group (EG) Within one RCC, an EG is a group of common calls, that can execute concurrently. Because a common call is equal to a logical transaction, a common call's transaction needs to own the replica's lock in order to execute. Therefore, an EG can consist either of only a single *writing* common call, or of any number of *reading* common calls. In the first case, no additional common calls can be registered in the EG. In the second case, only new reading common calls can be registered. In the second case however, a reading common call may need to be advanced to a writing common call. For one EG, this is allowed only to the first common call, that requests an advance and all subsequent requests within an EG cause the corresponding common call to be aborted. Obviously, the EG must ensure that such a ed writing common call is not allowed to execute until all other (reading) common calls finish. Also, an EG with a ed writing common call will refuse registration of any additional common calls.

Replica Common Call Context (RCCC) A common call within an EG of a certain replica (RCC) is represented by a RCCC. Until now, decisions regarding the execution permit have been based solely on the call's common call. Now, the task of RCCC is to synchronize single calls belonging to one common call. Each RCCC uses `CallInfoNode` objects to represent single calls (see figure 3.14). By being notified about each call's phase changes, a RCCC stores information about the state of every single call. It maintains a pointer to the currently active call (the only one allowed to execute), a queue for waiting calls, and a stack for suspended calls.

Because a common call in fact equals to a logical transaction, it is

straightforward the RCCC should be disposed when the transaction finishes. To accomplish this, the `ReplicaCommonCallContext` class extends the abstract class `DisposableTxnResource` described in section 3.4.4. This approach disposes RCCCs and (eventually) EGs after a transaction finishes and releases all locks, which is necessary before allowing next EGs and RCCCs to execute.

The basic design of the call guard system has been described, however, many implementation details must have been omitted. The most important properties of the presented synchronization system are:

1. determinism (the same registration order implies the same execution order), and
2. a high level of parallelism for concurrently executing calls.

3.6.4 Deadlock Prevention and Detection

It is relatively easy to imagine a situation, where two concurrent transactions acquire replica locks in a way causing a deadlock. As discussed in section 2.3.1, three different approaches solve this problem – *deadlock prevention*, *deadlock detection*, and *deadlock avoidance*.

The last approach would impose additional requirements on the transaction processing, therefore only the first two techniques have been implemented.

Deadlock prevention is easily achieved by aborting every transaction of a call, that would need to wait for a lock. Practically, this can only happen in the call guard system in two cases. Either the EG of a call is not the first in queue and so must wait for another EG to finish, or the call is in the first EG, but requires an advance and thus must wait for common calls (transactions) in the same EG to finish.

Alternatively, it is searched for deadlocks at call registration time. If it is found out the current call would cause a deadlock, it is aborted. By using this approach, waiting for locks can be enabled without the risk of a deadlock situation. In the rest of this section, the deadlock detection mechanism is explained in detail.

When a call is registered, a new `CallInfoNode` is created (see figure 3.14). It does not only store important information about the call, but also maintains dependencies to other `CallInfoNode`s. A call is dependent on another call if:

- the first call is a direct or indirect parent of the second call, or
- the first call's EG is waiting in a queue for the second call's EG to finish, or
- both calls are in the same EG, but the first call requires a lock advance.

By defining these dependencies, it is possible to construct a directed graph of `CallInfoNode` objects, the so called *waits-for graph* [BHG87] (see also section 2.3.1). A cycle in the waits-for graph means a deadlock situation, that must be resolved by aborting any call in the cycle.

In this implementation, a *depth first search* is started for every registered call from its `CallInfoNode` along the dependencies. If the search returns to the same `CallInfoNode`, this means the new call would cause a deadlock situation and therefore must be aborted. This is the task of the `DeadlockDetector` shown in figure 3.14.

The deadlock detection mechanism just described works only locally and thus requires all relevant calls to be multicasted to all nodes. This, however, reduces the possibilities of read call optimizations (see section 3.6.5). For better optimization possibilities, a distributed deadlock detection mechanism would be necessary.

The approach to handle deadlocks (prevention or detection) is configurable.

3.6.5 Optimization of Read Call Processing

With active replication, read calls do not need to be replicated among all nodes. Local execution improves read call performance by eliminating network communication and also reduces the load on all other nodes.

In the prototype, the deadlock detection must be taken with care (see section 3.6.4). As the deadlock detection mechanism only uses local data to

make decisions, all calls that could eventually cause a deadlock must be registered. This, however, implies replicating read calls as well. Otherwise, a deadlock might not be detected as the situation in figure 3.15 illustrates.

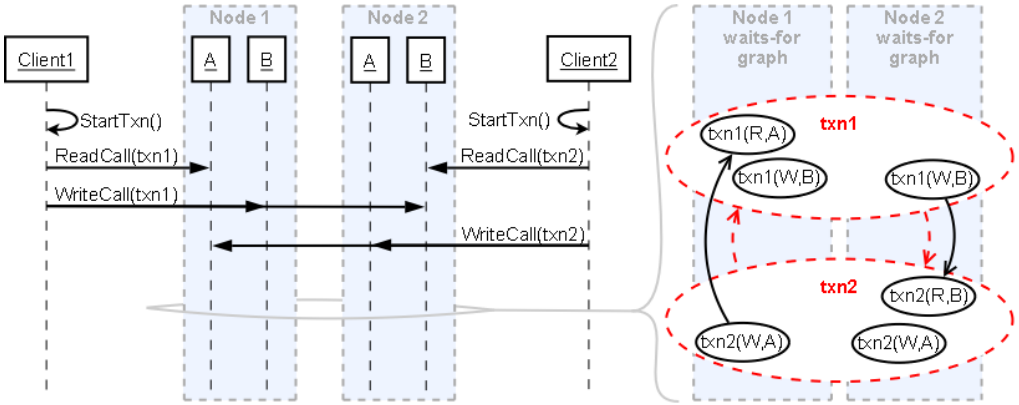

Figure 3.15: An undetected deadlock caused by local reads.

Two distinct clients start a transaction and place the subsequent calls within these transactions. *Client 1* (locally) reads replica *A* on *node 1*, while *client 2* (locally) reads replica *B* on *node 2*. Afterwards, *client 1* and *client 2* try to write replicas *B* and *A* respectively, within the previously started transactions. Now, the write calls are replicated to both nodes. Due to the locally executed read calls, each node does see a distinct call set. Therefore, *node 1* is aware of only one call dependency, while *node 2* of the another. Although the deadlock situation is obvious, none of the two nodes will detect it from its *waits-for graph*, because both are incomplete. By using distributed deadlock detection, a union of both graphs would be constructed and thus would contain the dependency cycle.

However, the situation just described is the only one causing this problem. If there is no (external) user supplied transaction with a top-level read call, no following write call can be started within the same transaction. If such a read call and all its (reading) subcalls never replicate to other nodes, the whole transaction will be known to exactly one node. This allows a local deadlock detection mechanism to detect all deadlocks.

Also consider a write call starting a reading subcall. Because every write call is replicated to all nodes, a locally started reading subcall will be

implicitly started on all nodes. Therefore, the transaction will appear the same on all nodes, which again allows proper deadlock detection.

To summarize the ideas just described a read call can be executed locally in the following cases:

- the local deadlock detection mechanism is not involved, or
- the read call is a top-level call and there is no (external) user supplied transaction, or
- the read call is a subcall.

This means, that only top-level read calls executing within an (external) user supplied transaction must be replicated. To allow local read call processing for all types of read calls, the deadlock detection would have to be distributed. However, it is not clear, whether this benefit would compensate the additional overhead of a distributed deadlock detection mechanism.

4 Evaluation

4.1 Hardware and Network Infrastructure

For the performance evaluation of the implemented prototype of the ARPPP a set of four computers has been connected to a local area network (LAN) using one router. The router is an AirPlus G 802.11g/2.4GHz Wireless Router. However, wireless connections have not been used and all computers have been plugged in using a network cable. The configuration is as follows:

1. *PC*
   - Intel Celeron CPU 2.80GHz
   - 556MB of RAM
   - HDD 80GB (7200rpm)
   - Fast Ethernet NIC (100Mbps)

2. *Laptop1*
   - Intel Pentium 4 CPU 2.00GHz
   - 228MB of RAM
   - HDD 30GB (4200rpm)
   - Fast Ethernet NIC (100Mbps)

3. *CF-Box*
   - VIA Nehemiah 664MHz
   - 556MB of RAM
   - Compact Flash (CF) memory 4GB
   - Fast Ethernet NIC (100Mbps)

4. *Laptop2*
   - Intel Celeron CPU 548MHz
   - 192MB of RAM
   - HDD 20GB (4200rpm)
   - 10/100 Ethernet Card (100Mbps)

The specified order also determines the computer group creation for testing. E.g., for testing with one node, PC (no.1) has been used. For testing with two nodes, PC (no.1) and Laptop1 (no.2) have been used, etc.

The CF-Box computer uses a compact flash (CF) memory in place of a hard disk. This type of memory is slower than the standard hard disk drives. Therefore, it is expected to have influence on the performance of write operations using a database backend.

4.2 Software Equipment

All four computers have been equipped with identical software. This includes the following:

- Microsoft Windows XP Service Pack 2 operating system,
- Microsoft .NET Framework 2.0,
- Spread 3.17.4 with identical network group configuration,
- MySQL 5.0 Server, and
- MySQL ODBC 3.51 Driver.

4.3 Replica Set for Testing

To allow a variety of performance tests, a set of four concrete replica classes has been designed as illustrated in figure 4.1.

All replica classes inherit the public field `x` from `AbstractReplica` and must implement the abstract method `WriteX_nested`. Implementations of this method write the local field `x` and subsequently call the `WriteX_nested` method on the referenced replica (if there is any).

At the program start, one instance of each class is created - called replica A, B, C, and D. Additionally, three constraints are defined:

- $A.x < 2000$ (critical/non-tradeable)
- $B.x < 2000$ (tradeable)
- $A.x + B.x < 3000$ (tradeable)

This design allows to place read, write, as well as nested calls. Due to the last constraint defined ($A.x + B.x < 3000$), all write operations on replicas A and B will cause in normal mode an additional nested read call, in order to validate this constraint.

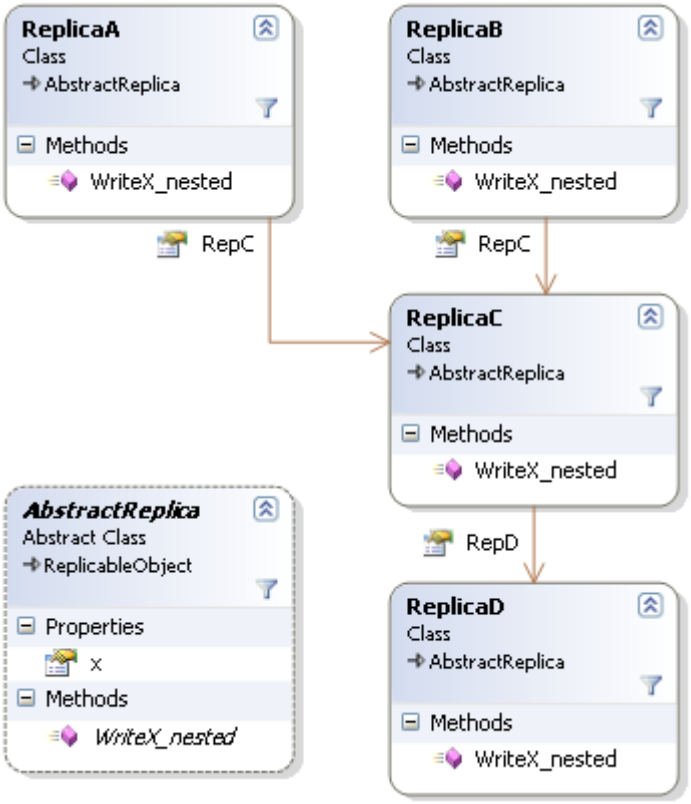

Figure 4.1: Replica set for testing.

4.4 Test Classes

For simple test creation, execution, and evaluation, the abstract class `AbstractTest` provides some common tasks to be inherited by concrete tests. Its class diagram is illustrated in figure 4.2.

`AbstractTest` requires its concrete subclasses to implement the abstract method `DoTestOperation`, which is then called multiple times in test iterations. To eventually modify its behaviour in time, this method gets the current iteration number as an input parameter.

The public field `NUM_OF_ITERATIONS` specifies how many iterations should be run to measure the operation performance. Using this information, the method `RunTest` calls `DoTestOperation` the given number of times and

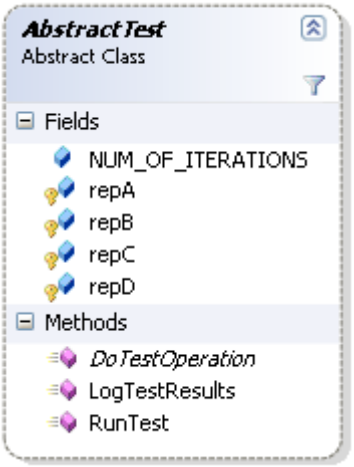

Figure 4.2: `AbstractTest` class diagram.

measures how long does it take. Afterwards, it is able to compute the average operation execution time. Both times are stored into private fields and are used by the `LogTestResults` method, which is called at the end of `RunTest`.

To eliminate the influence of any initializations or just-in-time compilations, the `RunTest` method runs one full operation iteration before the actual measuring starts.

Concrete test classes may override the `LogTestResults` method to log other test-specific results, too.

4.5 Performance Measurement

For all performance measurements, the tests have been run with `NUM_OF_ITERATIONS` equal to 1000. For every test, three separate runs have been performed and the average computed. The resulting diagrams show the time in milliseconds needed for a given operation to complete, where an operation can be a set of several calls.

Except of the first one, all tests have been started from the PC node.

4.5.1 Performance of the Nodes

To interpret the test results in context of hardware performance, first of all, the performance of simple read and write operations has been measured on each computer separately. The configured node count in this case has been 1 in all tests. Listings 4.1 and 4.2 show the particular test classes.

```
class Test_SimpleRead : AbstractTest
{
  ...
  public override void DoTestOperation(int iterNum)
  {
    int axVal = this.repA.x;
  }
}
```

Listing 4.1: Performance test of simple read call

```
class Test_SimpleWrite : AbstractTest
{
  ...
  public override void DoTestOperation(int iterNum)
  {
    this.repA.x = iterNum;
  }
}
```

Listing 4.2: Performance test of simple write call

Note that a write call on replica A causes a nested read call of B. Also, the simple write call test has been evaluated with two distinct system settings – with the database (DB) backend turned *on* and *off*. The diagram in figure 4.3 presents the corresponding results.

In this diagram, three operations have been measured. For every node, the first bar shows the performance of a simple read call (see listing 4.1). The second and third bar show a simple write call performance (see listing 4.2) with and without using a DB backend, respectively.

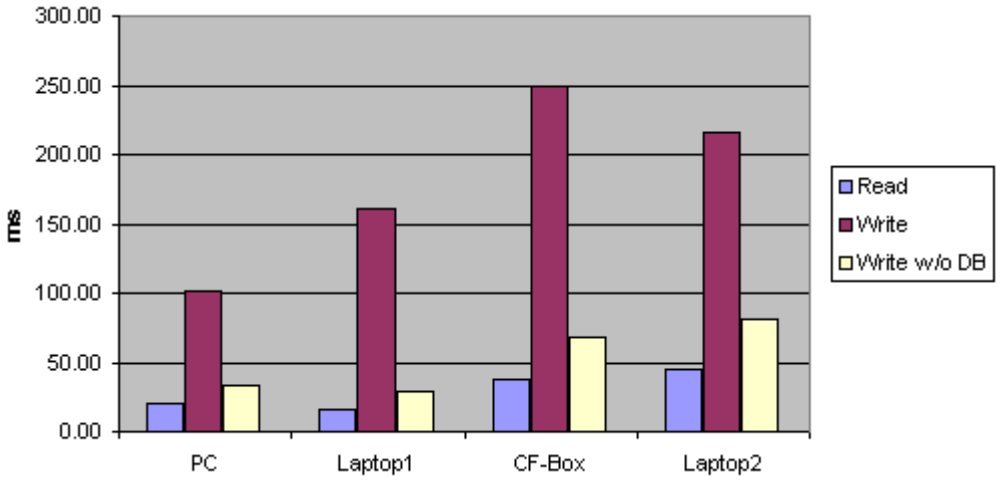

Figure 4.3: Simple call performance on separate nodes.

The results reflect the hardware performance of every node. For example, a write call with using DB backend on CF-Box takes the longest, because it has the slowest persistent memory.

4.5.2 Simple Calls

Here, the performance of simple calls (the same as in section 4.5.1) has been tested, however, now with different number of nodes in the system. The test has been started from the PC node. Figure 4.4 presents the test results.

Results indicate that the performance of write operations depends firstly on the performance of the slowest node in system, but secondly also on the number of nodes. With higher number of nodes, the overhead in group communication increases (this is partly visible also from read call performance) as well as the overhead of 2PC.

It can be observed, that the performance difference between a write call with and without using the DB backend is almost exactly equal to the same difference in the previous test (performance per node) for the slowest node in the system. So for example, in a system with three nodes, the performance difference between a write call using the DB backend and a write call not using the DB backend is 179 milliseconds (see figure 4.4). The

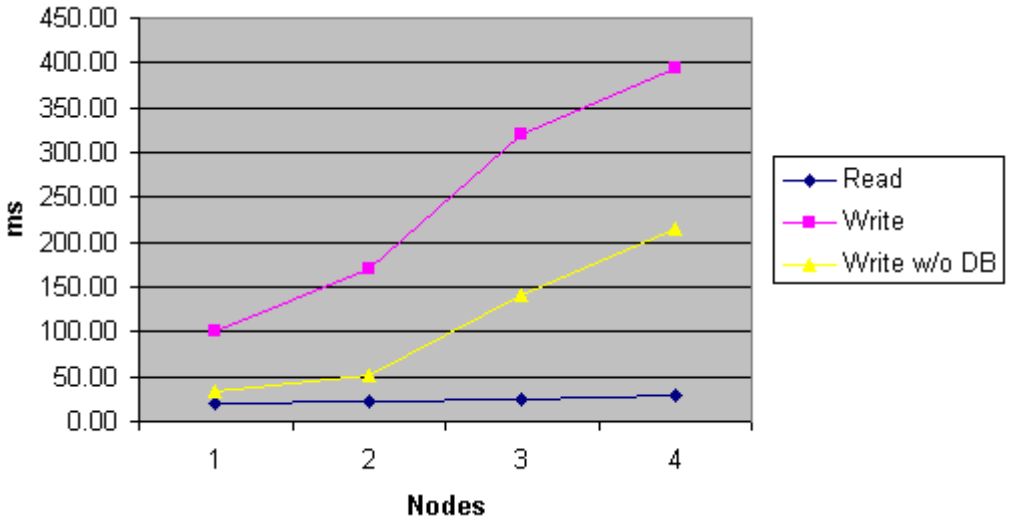

Figure 4.4: Simple call performance.

slowest node in this test set is the CF-Box node. The results of the very first test returned for the CF-Box (as a single system node) a difference between the same types of write calls equal to 180 milliseconds (see figure 4.3).

With an increasing node count, the simple read performance grows only very slowly due to the read call optimizations introduced in section 3.6.5. The slight growth relates to the group communication which slows down with the increasing number of nodes.

The influence of hardware on overall performance gets multiplied, because it affects not only the speed of call processing itself, but also the speed of group communication and transaction coordination. This is because every node must run its own instance of group communication service as well as the distributed transaction coordinator.

4.5.3 Nested vs. Serial Calls

In this test, a performance comparison between three nested and three serial write calls have been made. Listings 4.3 and 4.4 show the corresponding test classes.

```
class Test_NestedSerial_Nested : AbstractTest
{
  ...
  public override void DoTestOperation(int iterNum)
  {
    this.repA.WriteX_nested(iterNum);
  }
}
```

Listing 4.3: Performance test of three nested write calls

```
class Test_NestedSerial_Serial : AbstractTest
{
  ...
  public override void DoTestOperation(int iterNum)
  {
    using (TransactionScope ts = new TransactionScope())
    {
      this.repA.x = iterNum;
      this.repC.x = iterNum;
      this.repD.x = iterNum;
      ts.Complete();
    }
  }
}
```

Listing 4.4: Performance test of three single write calls

In the second test class, the single write calls are grouped into one transaction as the nested write calls also execute within one transaction. In the first case, three nested write operations are executed and then the replica B is read in order to validate the constraint between A and B. In the second case, replica A is written, then B is read, and finally two other replicas are written. This difference in call order, however, is not expected to have influence on the results.

The outcome of this test is presented in figure 4.5.

The diagram shows, that there is not really a difference between the two tested operations. I.e. the performance of same calls within one transaction

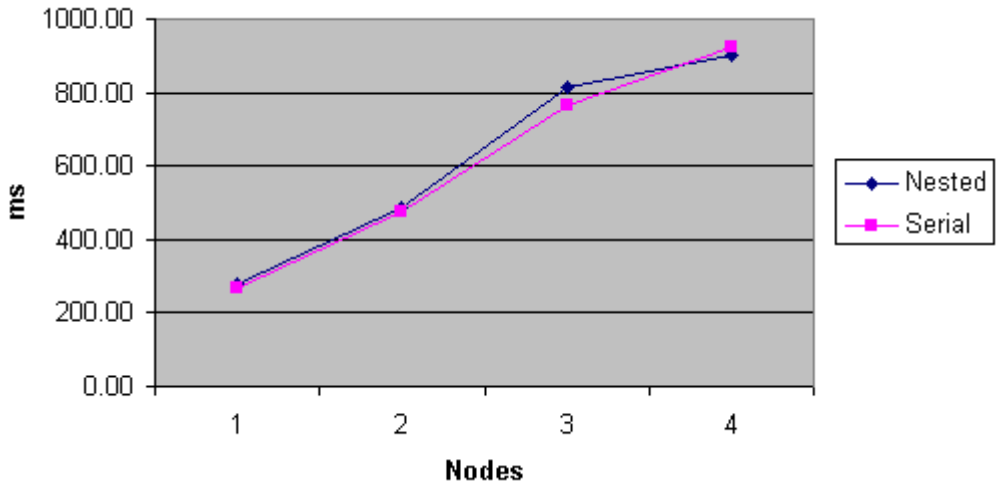

Figure 4.5: Nested vs. serial call execution.

does not depend on whether these calls are nested or executed separately in a row.

4.5.4 Transaction Grouped vs. Ungrouped Calls

In the last test, the performance of five calls has been compared – once if grouped into a transaction and secondly if not grouped into a transaction. Listings 4.5 and 4.6 show the test classes.

```
class Test_TxnCallGroup_Grouped : AbstractTest
{
  ...
  public override void DoTestOperation(int iterNum)
  {
    using (TransactionScope ts = new TransactionScope())
    {
      this.repD.x = this.repC.x;
      this.repC.x = this.repA.x;
      this.repA.x = iterNum;
      ts.Complete();
    }
  }
```

```
}
```

Listing 4.5: Performance test of five calls grouped into a transaction

```
class Test_TxnCallGroup_Ungrouped : AbstractTest
{
  ...
  public override void DoTestOperation(int iterNum)
  {
    this.repD.x = this.repC.x;
    this.repC.x = this.repA.x;
    this.repA.x = iterNum;
  }
}
```

Listing 4.6: Performance test of five calls NOT grouped into a transaction

The call order is the same in both cases. Replica C is read in order to write replica D. Then, A is read in order to write C and finally A is written. As already discussed, writing A causes another read of B.

There are two contradictory influences between these operations. The read calls to replica C and A are processed in a distinct manner, because of the read call optimizations (see section 3.6.5). A read call within an external (user supplied) transaction must be replicated to all nodes, while a read call without such a transaction can be optimized and executed only locally.

The second point is, that in case of calls *not* grouped into a transaction, a transaction must be started and finished five times, while for grouped calls this happens only once.

Figure 4.6 shows the results of this test.

Although the differences in performance of both operations are not big, they most probably refer to the issues described earlier. For a single node, the performance of both operations appears to be the same. In this case, the communication overhead in the group communication and transaction coordination is eliminated. Furthermore, in this case there is obviously no difference in processing read calls.

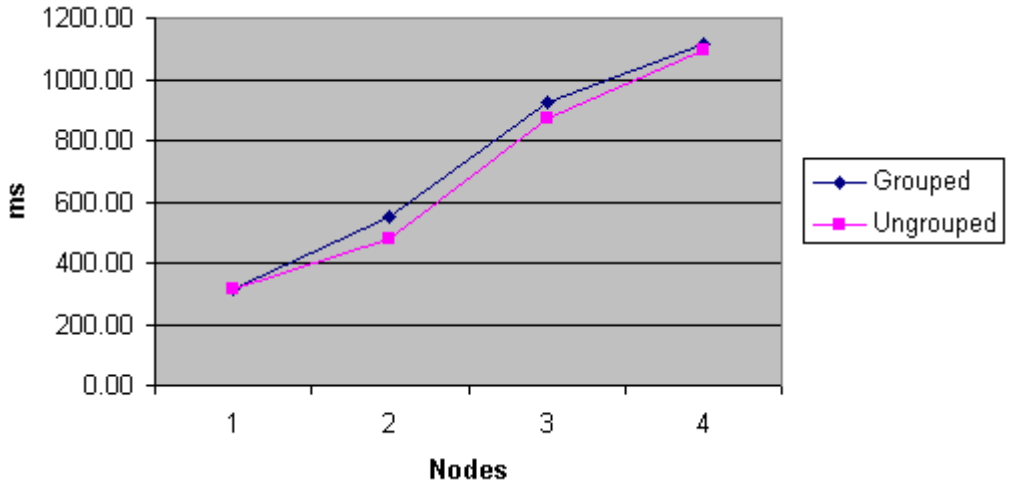

Figure 4.6: Calls grouped vs. ungrouped into a transaction.

For more than one node in the system, the calls not grouped into a transaction achieve slightly higher performance. This corresponds to the fact, that the read calls are allowed to execute locally. With an increasing number of nodes, however, the gap between performance of grouped and ungrouped operations decreases. This might be caused by network communication overhead in group communication and transaction coordination increasing faster than the overhead of replicated read calls.

4.6 Transaction Abort Rates

One of the goals of this thesis was to introduce transactional support for operations. As already shown, it has been realized using the `System.Transactions` namespace of .NET Framework 2.0. This allows, for the first time, to evaluate the behaviour of ACBRM for concurrent operations in a .NET environment. For this purpose, transaction abort rates have been tested for several operation types.

All the tests have been performed in a system consisting of four nodes, where two clients concurrently run a certain operation. Both clients have run the operation 500 times (`NUM_OF_ITERATIONS` equal to 500), while it has been recorded how many operations have finished successfully.

All operation types have been tested with *deadlock detection* as well as *deadlock prevention*. This also allows a comparison of these two deadlock handling mechanisms. Deadlock detection only causes a transaction abort if a deadlock is detected. On the other hand, using deadlock prevention will abort the transaction every time a lock cannot be obtained.

The client operations have been run on the PC and Laptop1 computers.

4.6.1 Tested Operation Types

For the evaluation of transaction abort rates, four operation types have been chosen:

1. simple read call,
2. simple write call,
3. simple write call (with mutual constraint checking), and
4. nested write call (with mutual constraint checking).

A generic test class is presented in listing 4.7.

```
class Test_TxnAbortRate_??? : AbstractTest
{
  ...
  public override void DoTestOperation(int iterNum)
  {
    try
    {
      ...perform the operation...
    }
    catch { ...record the aborted transaction... }
  }
}
```

Listing 4.7: A general transaction abort rate test

The operation to be tested is enclosed in a *try-catch* block, so if the operation is aborted (throws an exception), this can be recorded and the transaction abort rate computed.

In the rest of this section, each operation type is described. Also, the exact operation run by each of the two concurrent clients is stated.

Simple Read Call

- `int ax = this.repA.x;` (for both clients the same)

In this test case, both clients have repeated the same operation – a simple read call to replica A. For both deadlock handling mechanisms, the transaction abort rate is expected to be 0% as two read locks never conflict.

Simple Write Call

- `this.repA.x = iter;` (for both clients the same)

A simple write call on replica A has been repeated on both clients in the second test case. Due to automatic constraint checking, this write call causes an additional read call to replica B within the same transaction.

Because transactions run by both clients obtain locks in the same order, a deadlock can never occur. Therefore, if using deadlock detection, the operations should never be aborted and the transaction abort rate should be 0%. Otherwise, if deadlock prevention is used, a running transaction owning a write lock on A will cause all other transactions trying to acquire the same lock to abort.

Simple Write Call with Mutual Constraint Checking

- `this.repA.x = iter;`
- `this.repB.x = iter;`

This testcase is similar to the previous one, however, while the first client is writing replica A, the second is writing replica B. Because of a constraint defined between A and B, both write operations imply a read call to the other replica.

This situation can easily cause a deadlock, as both operations acquire conflicting locks on the same replicas, but in distinct order. Therefore, when using deadlock detection, transactions may be aborted, in order to resolve deadlock situations. Using deadlock prevention causes abortions every time a transaction cannot acquire a lock, because a conflicting lock is held by another transaction.

Nested Write Call with Mutual Constraint Checking

- `this.repA.WriteX_nested(iter);`
- `this.repB.WriteX_nested(iter);`

In the last test case, the transaction abort rate of nested operations has been tested. One of the clients starts the nested write on replica A, while the other one on replica B. In both variants, this again causes an additional read call to verify the constraint between A and B.

Similar to the previous test case, positive transaction abort rates are expected for both deadlock handling mechanisms.

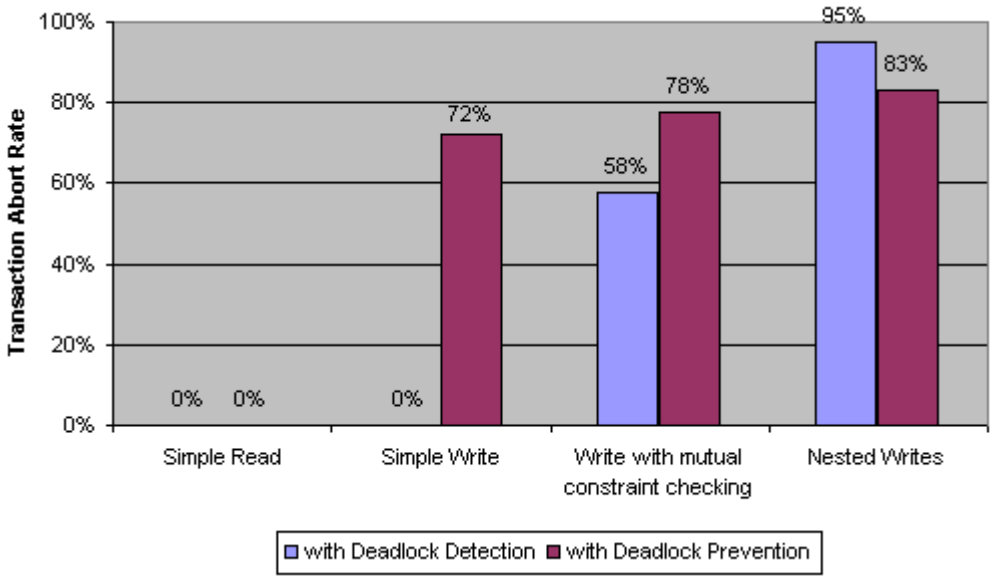

Figure 4.7: Transaction abort rates for different operation types.

4.6.2 Test Results

Figure 4.7 presents a result summary of the transaction abort rate tests.

As expected, two concurrent read calls accessing the same replica do not conflict and thus do not need to be aborted.

Although, two concurrent write calls conflict on the same replica, they can be serialized without causing a deadlock. In case deadlock detection is used, calls are delayed as necessary and none of them must be aborted. When using deadlock prevention, delaying a call is not allowed and such call must be aborted instead. With deadlock prevention the transaction abort rate depends on how long a transaction blocks a certain lock causing a conflict with concurrent transactions. In this case, a write lock is blocked for the duration of one write call and one read call (to validate a constraint). Also only one write-write conflict between two transactions may arise.

In the third test case, simple write operations are started on distinct replicas causing a mutual constraint check. As explained earlier, this can cause deadlock situations so transactions may need to be aborted, also when using

deadlock detection. A transaction abort rate somewhat above 50% indicates, that the time window between acquiring the write lock and acquiring the read lock is something longer than the time span from transaction start (actually end of the previous transaction) and acquiring the write lock.
Using deadlock prevention has a similar effect than in the previous case. Although, a lock conflict with respect to one replica is less probable to occur (one of the "collision windows" is shorter), the abort rate is slightly higher because there are read-write conflicts possible on two replicas.

In the last case, the two concurrent clients start a nested call, however, each on a different replica. So, one client runs transactions of the form $W(A) \rightarrow W(C) \rightarrow W(D) \rightarrow R(B)$, while the second client runs transactions of the form $W(B) \rightarrow W(C) \rightarrow W(D) \rightarrow R(A)$. Using deadlock detection, transactions get aborted due to deadlock situations arisen from acquiring conflicting locks on replicas A and B in a distinct order. Because the "collision window" here is relatively long, the transaction abort rate in this case is appropriately (very) high.
With deadlock prevention, the primary reason for transaction abortion arises from the write-write conflict when obtaining a lock on replica C. A relatively long blocking of the write lock on C causes appropriately high abort rates. There is also an influence of the read-write conflict on replicas A and B, however, it is much smaller, because it extends the time span of possible lock blocking in a small extent.
As it is apparent from the results in figure 4.7, the "collision window" for deadlocks (when using deadlock detection) is so long, that the transaction abort rate in this case is even higher than if using deadlock prevention.

5 Summary and Conclusions

5.1 Summary

A new class of replication protocols has been recently introduced, allowing an adaptable trade-off between constraint consistency and availability. The key idea is to allow tradeable constraints to be temporarily violated during degraded situations, so that replicas not affecting critical (non-tradeable) constraints stay available in case of node failures or network partitions.

This thesis presents a prototype implementation of a concrete protocol called *Active Replication Per Partition Protocol (ARPPP)*, which integrates the above-mentioned trade-off into the active replication scheme. The developed prototype based on .NET supports nested replica invocations as well as distributed transactions. A group communication toolkit provides the necessary mechanisms needed for inter-node communication.

Evaluation of the developed prototype is part of the thesis as well. Not only the performance of simple read and write calls has been measured, but also performance of nested calls and calls grouped into transactions. What more, thanks to the full transaction support, concurrency behaviour has been also tested by measuring transaction abort rates for different combinations of concurrent operations.

5.2 Conclusions

The prototype has been implemented to the planned extent. Compared to traditional active replication, the biggest overhead in ARPPP is caused by additional read calls needed for constraint validation. However, due to the optimization of read call processing, this overhead is kept relatively small.

It turned out to be possible to integrate the transaction mechanisms provided by .NET Framewrok 2.0 into the prototype. Replicas exposed by the system behave as transaction resource managers in context of .NET. This allows the client to combine transaction operations on other heterogeneous resources with calls to the system and to externally manage the transaction.

To allow the usage of nested calls, a special synchronization mechanism had to be developed. The possibility of inter-replica references and nested calls increases the variability of applications to a considerable extent.

Evaluation of the prototype showed that the performance of all replicated operations is influenced by several aspects. Firstly, it is in direct proportion to the slowest node in the system. Also, with growing number of nodes, the group communication overhead increases as well as the transaction coordination overhead does. The execution of nested calls and calls grouped to logical transactions has not proven any significant overhead in the performance. Optimized read calls perform much better as they are executed only locally.

Among all protocols supporting the trade-off between constraint consistency and availability, this implementation of ARPPP is the first one supporting transactions in a .NET environment. Therefore, the evaluation of concurrent operations behaviour has been possible. It has shown, that the read calls required for constraint validation may cause several issues. These additional operations belong to the same transaction as the originating write call and thus extend its duration, so that locks are blocked for a longer period of time. Even worse is the impact on the probability of deadlocks. Concurrent write operations on two replicas affecting a common constraint are likely to cause deadlocks. If such write calls also initiate nested calls, the probability of a deadlock can climb up very high.

The evaluation results indicate, that the necessity of constraint validation implies a performance decrease and higher transaction abort rates of (concurrent) operations. Therefore, the constraint definition in a particular application should always be done carefully and with respect to the performance.

5.3 Future Work

This thesis primarily concentrates on the replication protocol itself. Implementations of the reconciliation service and the system of constraints are very basic, providing only the features necessary for the prototype and its evaluation. Although fully functional, the performance of the reconciliation algorithm could be improved considerably.

Furthermore, a naming and lookup service would be of interest.

To improve performance in case of a high replica number installed, the activation service should be able to passivate unused replicas and reactivate them as necessary.

According to the evaluation measurements, optimized read calls, although executed locally, slightly lose on performance with growing number of nodes in the system. This is because of the way an addressed message is delivered by the group channel. In practice, the message is broadcasted to all nodes, but only the addressee accepts it. This behaviour is subject to optimization. Either a direct delivery within the group should be implemented, or a TCP channel should be used instead of the group channel.

The evaluation of the ARPPP prototype has shown high transaction abort rates for concurrent operations consisting of nested calls and a mutual constraint validation (see section 4.6). To lower the abort rates, some measures could be implemented and tested in future versions. The conservative two-phase locking (2PL) approach would avoid deadlocks completely and thus would ensure no transaction abort rates at all. However, it would place extensive restrictions on the application flexibility. A softer option would be to use conservative 2PL only for a single write call. I.e. the replicas needed for constraint validation would be locked for read (in a predefined order), before the update operation is executed.

Another approach to lower the transaction abort rate would be to separate the operation execution and constraint validation actions, so that each of them is performed within its own transaction. In this case, however, some kind of constraint consistency re-establishment (e.g. compensation actions) must be used for resolving any conflicts found during constraint validation.

5.4 Related Work

For a long time, improving availability of distributed systems by relaxing consistency has been focused on optimistic approaches providing no consistency guarantees at all.

[YV02] has introduced a consistency model where consistency is expressed by so-called *conits* (logical consistency units). This model has also been

used in the implementation of the TACT (Tunable Availability and Consistency Trade-offs) middleware layer. To quantify conits, three application-independent metrics have been proposed – *numerical error*, *order error*, and *staleness*. "Numerical error limits the total weight of writes on a conit that can be applied globally across all replicas before being propagated to a given local replica. Order error limits the number of tentative writes on a conit (subject to reordering) that can be outstanding at any one replica, and staleness places a real-time bound on the delay of write propagation among replicas." [YV02] By defining application-specific bounds using conits, TACT provides a tunable trade-off between system availability and replica consistency. However, in contrast to the subject of this thesis, it does not consider constraint consistency.

Smeikal analyses in his PhD thesis [Sme04] three types of consistency – replica consistency, concurrency consistency, and constraint consistency. It suggests further constraint consistency as the system correctness criterion to be traded against availability. This idea has been further developed in frame of the DeDiSys project and has resulted into the generic ACBRM replication model [OFG07]. Such replication techniques for balancing data integrity with availability are discussed in detail in Osrael's PhD thesis [Osr07].

Apart from the ARPPP as the subject of this Master's thesis, two other protocols based on the ACBRM have been described and implemented. The one is the *Primary-per-Partition Protocol (P4)* [BBG+06], based on the primary-backup scheme, while the other is *Adaptive Voting (AV)* [OFGG06], based on the quorum-consensus approach.

All three implementations integrate its components into the .NET architecture introduced by [Hab06]. This is a mapping from a platform-independent system architecture [OFG+06] worked out within the DeDiSys project.

Chlaupek's Master's thesis [Chl07] focuses on implementation and evaluation of the Adaptive Voting protocol in a .NET environment. Implementation of the Primary-Backup and Primary-per-Partition replication protocols is the subject of Weigl's Master's thesis [Wei06], whereas Stoifl [Sto06] presents an implementation of the Quorum Consensus replication protocol in a .NET environment. All three implementations rely on the Spread communication toolkit to provide a group communication service. [Sto06] also compares the features of .NET Remoting and Spread in detail.

In contrast to the ARPPP implementation presented in the given thesis, these works do not support nested transactions.

Moser [Mos07] analyses different reconciliation protocols with respect to the replica consistency as well as constraint consistency. For his experimental evaluation he provides a .NET-based test framework, which, in contrast to the given thesis, is focused on reconciliation only.

List of Figures

Listings

References

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan.-March 2004.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10. ACM, 1995.

[BBG+06] Stefan Beyer, Mari-Carmen Bauls, Pablo Galdmez, Johannes Osrael, and Francesc D. Muoz-Esco. Increasing availability in a replicated partitionable distributed object system. In *ISPA*, volume 4330 of *Lecture Notes in Computer Science*, pages 682–695. Springer, 2006.

[BG81] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[BHG87] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[BMST93] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In *Distributed systems (2nd Ed.)*, pages 199–216, New York, NY, USA, 1993. ACM Press/Addison-Wesley Publishing Co.

[Chl07] N. Chlaupek. Implementation and evaluation of the adaptive voting replication protocol in a .net environment. Master's thesis, University of Applied Sciencies, FH Campus Wien, 2007.

[DB85] Dančo Davčev and Walter A. Burkhard. Consistency and recovery control for replicated files. *SIGOPS Oper. Syst. Rev., ACM*, 19(5):87–96, 1985.

[DGS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Comput. Surv.*, 17(3):341–370, 1985.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[FS01] P. Felber and A. Schiper. Optimistic active replication. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 333–341. IEEE Computer Society, Apr 2001.

[GA87] H. Garcia-Molina and R.K. Abbott. Reliable distributed database management. *Proceedings of the IEEE*, 75(5):601–620, May 1987.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[GHR97] Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham. Revisiting commit processing in distributed database systems. *ACM SIGMOD Rec.*, 26(2):486–497, 1997.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM.

[GL06] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[GS97] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer, IEEE Computer Society Press*, 30(4):68–74, Apr 1997.

[Hab06] Igor Habjan (ed.). Software prototype and refined design. Technical Report D3.3.2, DeDiSys Consortium (www.dedisys.org), December 2006.

[JM87] Sushil Jajodia and David Mutchler. Enhancements to the voting algorithm. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases*, pages 399–406, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[JPA00] Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Sergio Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *SRDS '00: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, page 164, Washington, DC, USA, 2000. IEEE Computer Society.

[JSSG05] M. Jandl, A. Szep, R. Smeikal, and K. M. Goeschka. Increasing availability by sacrificing data integrity - a problem statement. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 9*, page 291.3. IEEE Computer Society, 2005.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Laz08] Florin Lazar. Florin lazar - transactions. Florin Lazar's blog (on MSDN), December 2003 - February 2008. http://blogs.msdn.com/florinlazar/.

[LFT02] Deron Liang, Chen-Liang Fang, and JiChiang Tsai. A nested invocation suppression framework for active replication fault-tolerant corba. In *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, pages 757–762. IEEE Computer Society, 2002.

[Low05] Juval Lowy. Introducing system.transactions in the .net framework 2.0. Microsoft Development Network (MSDN), December 2005. http://msdn2.microsoft.com/en-us/library/ms973865.aspx.

[LXML07] Lingxia Liu, Jingbo Xia, Zhiqiang Ma, and Ruixin Li. Rapid-response replication: A fault tolerant algorithm based on active replication. In *International Conference on Computational Science (3)*, pages 133–136. Springer, 2007.

[Mos81] E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.

[Mos07] Christian Moser. Analysis of reconciliation protocols for divergent replicas. Master's thesis, Vienna University of Technology, 2007.

[OFG+06] J. Osrael, L. Froihofer, K. M. Goeschka, S. Beyer, P. Galdamez, and F. Munoz. A system architecture for enhanced availability of tightly coupled distributed systems. In *ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security*, pages 400–407, Washington, DC, USA, 2006. IEEE Computer Society.

[OFG07] J. Osrael, L. Froihofer, and K.M. Goeschka. Availability/consistency balancing replication model. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 26-30 March 2007.

[OFGG06] Johannes Osrael, Lorenz Froihofer, Matthias Gladt, and Karl M. Göschka. Adaptive voting for balancing data integrity with availability. In *OTM Workshops (2)*, pages 1510–1519. Springer, 2006.

[OFKG05] Johannes Osrael, Lorenz Froihofer, Hubert Kuenig, and Karl M. Goeschka. Scenarios for increasing availability by relaxing data integrity. In *Innovation and the Knowledge Economy: Issues, Applications, Case Studies*, pages 1396–1403. IOS Press, October 2005.

[OFS+06] Johannes Osrael, Lorenz Froihofer, Georg Stoifl, Lucas Weigl, Klemen Zagar, Igor Habjan, and Karl M. Goeschka. Using replication to build highly available .net applications. In *DEXA '06: Proceedings of the 17th International Conference on Database and Expert Systems Applications*, pages 385–389, Washington, DC, USA, 2006. IEEE Computer Society.

[Osr07] Johannes Osrael. *Replication Techniques for Balancing Data Integrity with Availability.* PhD thesis, Vienna University of Technology, 2007.

[Pec06] Jan Peciva. Active transaction approach for collaborative virtual environments. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pages 171–178. ACM, 2006.

[PL88] J.-F. Paris and D.D.E. Long. Efficient dynamic voting algorithms. In *Data Engineering, 1988. Proceedings. Fourth International Conference on*, pages 268–275. IEEE Computer Society, 1-5 Feb 1988.

[PL91] J.-F. Paris and D.D.E. Long. Voting with regenerable volatile witnesses. In *Data Engineering, 1991. Proceedings. Seventh International Conference on*, pages 112–119. IEEE Computer Society, 8-12 Apr 1991.

[RSL78] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, 1978.

[Sch93] Fred B. Schneider. Replication management using the state-machine approach. In *Distributed systems (2nd Ed.)*, 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, New York, NY, USA, 1981. ACM.

[Sme04] Robert Smeikal. *Trading Consistency for Availability in a Replicated System.* PhD thesis, Vienna University of Technology, June 2004.

[SNM85] Mukul K. Sinha, P. D. Nandikar, and S. L. Mehndiratta. Timestamp based certification schemes for transactions in distributed database systems. *ACM SIGMOD Rec.*, 14(4):402–411, 1985.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[Sto06] Georg Stoifl. Implementierung des quorum consensus replikationsprotokolls in einer .net umgebung basierend auf gruppenkommunikation. Master's thesis, University of Applied Sciences Technikum Wien, 2006.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[TS02] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, 2002.

[Wei06] Lucas Weigl. Implementierung von replikations-protokollen in der .net umgebung. Master's thesis, University of Applied Sciences Technikum Wien, 2006.

[WPS+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. *IEEE ICDCS: Int. Conf. Distributed Computing Systems, 2000. Proceedings.*, pages 464–474, 2000.

[WS92] Gerhard Weikum and Hans-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann, 1992.

[YV02] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.

Appendix

Installation Guide

Pre-requisites

The following software is necessary in order to run the program:

- Microsoft Windows operating system
- .NET Framework 2.0 or higher
- Spread 3.17.4
- Database with an ODBC connector available

No special hardware requirements must be met. Note, that the exact version of Spread must be installed, as higher versions are not compatible.

If the program is going to be tested on two or more computers, the software listed must be installed on each of them. Optionally, one database can be shared among several computers (it can also be located on a dedicated computer). All computers must be connected to a common network.

Enabling Transactions Across Computers

As the Windows XP Service Pack 2 disables network access for the Distributed Transaction Coordinator (DTC), its settings should be checked and eventually modified in order to enable transactions across computers. Except of enabling network transactions, the firewall should be configured not to block the DTC service. A detailed description of how to do this can be found in Florin Lazar's blog article "XP SP2 and Transactions" (*http://blogs.msdn.com/florinlazar/archive/2004/06/18/159127.aspx*).

Spread Configuration

The Spread's configuration file (`spread.conf`) must be edited in order to define the computers forming a group, i.e. a *Spread Segment*.

If all system nodes will be run on a single computer, then the default configuration is sufficient:

```
Spread_Segment 127.0.0.255:4803 {
   localhost  127.0.0.1
}
```

Otherwise, the spread segment must be edited to list all computers belonging to the communication group. For example, the configuration used during evaluation of this prototype was the following:

```
Spread_Segment 192.168.0.255:4803 {
   oprsal-1     192.168.0.101
   cf-box       192.168.0.103
   gericomntbk  192.168.0.201
   deniskantbk  192.168.0.202
}
```

Database Configuration

Any database can be used if there exists the appropriate ODBC Driver for it. After installing the ODBC driver, a Data Source Name (DSN) for the database must be created and configured. During evaluation, a MySQL database with the MySQL ODBC 3.51 Driver was used. How to configure an ODBC connector and set a DSN can be found at *http://dev.mysql.com/doc/refman/5.0/en/myodbc-configuration-dsn-windows.html*

Running the Program

To start a new node, a Spread instance must be running on the local computer. A new node is started by running the `DeDiSys-Test.exe` executable. The following commandline arguments are accepted:

- `n:<node_count>` - Number of nodes in a healthy system. Default is 4.
- `useDbBackend:[true|false]` - Whether to use a DB backend. Default is `false`.
- `dbDsn:<dsn>` - Data Source Name of the DB (only if using DB).
- `dbName:<db_name>` - Name of the DB (only if using DB).
- `dbUser:<db_user>` - Name of the DB user (only if using DB).
- `dbPassword:<db_passwd>` - Password for the specified user (only if using DB).
- `allowCallDelay:[true|false]` - Whether to use deadlock detection (true), or deadlock prevention (false). Default is `true`.

The following example will start a node, that expects 3 nodes in a healthy system and that uses a database backend with the DSN "MySQL DB", database "dedisys_db_1", user "root", and password "123":

```
DeDiSys-Test.exe n:3 dbDsn:"MySQL DB" dbName:dedisys_db_1
dbUser:root dbPassword:123
```

By default, the node will use deadlock detection and thus allow call delaying for concurrent calls.

All commands available can be viewed by typing '?' or 'help'.

When running any of the tests, the test results are stored in a file called `DeDiSysTest.log`.

Logging

Logging is done by using the *log4net* library. The log output can be configured in the standard `DeDiSys-Test.exe.config` file.

Note, that all logging actions decrease performance considerably, so no output should be produced during performance evaluation.