DIPLOMARBEIT

# Evaluation of Partitionable Replication Protocol Improvements in an Enterprise JavaBeans Environment

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs unter der Leitung von

Univ.-Prof. Dr. Schahram Dustdar
und
Dr. Karl M. Göschka,
Dipl.-Ing. Lorenz Froihofer
als verantwortlich mitwirkende Assistenten am

Institutsnummer: 184-1
Institut für Informationssysteme
Arbeitsbereich für Verteilte Systeme

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Dominik Ertl Bakk.tech.
Matr.Nr. 0030055
Jägerhausgasse 41, 1120 Wien

Wien, im Oktober 2007

**Abstract** Replication is often used to provide fault tolerance for better availability in case of node and link failures. Link failures may lead to network partitions that have to be addressed by partitionable replication protocols. In the past, a primary-per-partition protocol has been implemented for Enterprise JavaBeans within the Dependable Distributed Systems research project. Within this master's thesis, an enhanced version of this protocol is designed and implemented based on the primary-per-partition approach in combination with a centralized configurable history service for logging states and operations. This work further focuses on the reconciliation phase where the different partitions have to be made consistent again. The new replication protocol is examined and compared to the given primary-per-partition protocol. It is shown that the propagation of replicas occurs clearly faster with the new protocol. Additionally, the re-establishment of constraint consistency is implemented in two different ways. Both mechanisms perform well and need nearly the same amount of time to provide constraint consistency again.

**Kurzfassung** Replizierung von Daten ist eine weit verbreitete Technik, um Fehlertoleranz für bessere Verfügbarkeit im Falle von System- und Verbindungsfehlern zu erreichen. Verbindungsfehler können zu Netzwerkpartitionen führen, die von sogenannten partitionierbare Replikationsprotokollen gehandhabt werden. In früheren Arbeiten wurde im Rahmen des Forschungsprojektes “Dependable Distributed Systems” ein “Primary-per-partition” Protokoll für die Enterprise JavaBeans Umgebung entwickelt. Das Ziel dieser Diplomarbeit ist das Design und die Implementierung eines Replikationsprotokolls, welches auf diesem “Primary-per-Partition” Ansatz basiert. Dazu ist auch ein zentral konfigurierbares History Service entwickelt worden, welches dem Speichern von Zuständen und Operationen in der zugrunde liegenden Middleware dient. Diese Arbeit fokussiert auf die “Reconciliation” Phase, in der verschiedene, vorher getrennte, Partitionen wieder zusammengeführt werden um einen systemweit konsistenten Zustand zu erlangen. Dieses neue Replikationsprotokoll ist geprüft und mit der vorherigen Lösung verglichen worden. Es zeigt sich, dass die Verteilung von Replikaobjekten mit dem neuen Protokoll deutlich schneller funktioniert. Das Erreichen von “Constraint Consistency” mittels des Protokolls ist auf zwei verschiedene Arten implementiert. Beide Mechanismen haben eine erwartet gute Performance und benötigen eine vergleichbare Zeitspanne um Constraint Consistency wieder herzustellen.

# Acknowledgements

At first I give special thanks to my beloved Stefanie, my whole family and friends out there — which unfortunately are distributed like the systems described in this thesis.

Built on the shoulder of giants: This is not just a dumb sentence when speeches are held during the Oscar Academy Awards ceremony. It is the truth, whenever one pokes his nose into a (technology) research field and tries to contribute to it. Thus, special thanks go to Dr. Karl M. Göschka who gave me the possibility to participate in this interesting research project. His impressive didactic skills and knowledge regarding many areas within the information technology will truly stay in my memories. Thanks also to Dipl.-Ing. Lorenz Froihofer - with his patience and insistently asserted inputs he kept the thesis on track. I am grateful to Mag. Theresa Klimpfinger for the review of the textual version.

Last but not least I would like to thank soccer and everything that relates to it. It gave me hours of relaxation after a day's hard work.

# Contents

# List of Figures

## Listings

# 1 Introduction

Flowery language and enthralling tales are typically not the focus of a master's thesis. But a little story shall provide the reader of this document with background information, the "what is it all about?" behind the exhaustive title "Implementation and Evaluation of Partitionable Replication Protocols in an Enterprise JavaBeans environment":

Let me introduce Ulrich: Ulrich is a soccer enthusiast and does not miss any game of the Austrian national soccer team. Usually, he buys the tickets for the games weeks before the match-up starts. And he is a very friendly fan, so he buys tickets not only for himself, but also for all his friends and family members who want to have a great time in the stadium. Unfortunately, there are many people like Ulrich who want to attend the games, therefore the Austrian Football Association limits the sale of tickets to a maximum of four tickets per person for one match in a stadium. This limit is called a "constraint". There are a number of ticket agencies in his hometown, and all these ticket agencies are connected together. Such a connected system is called a "tightly coupled distributed system". So Ulrich may buy two tickets for his brother and himself in agency A in the morning, and, after having called two friends in the afternoon, he may buy two tickets in agency B on the next day (it is nearer to his working place). Luckily for the Austrian Football Association, the computer systems stay connected during the two purchases. After the second acquisition, the sales person informs Ulrich that he is not allowed to buy any further tickets in the future for this match.

But what happens if - let us say due to a careless digger driver - the telephone and network cable of agency B has been cut shortly after Ulrichs first buy? Agency A and agency B each have saved the information that Ulrich has bought two tickets, and due to the constraint is allowed to buy two more tickets. If Ulrich had astute intentions, he could now buy two more tickets in agency A again, run to agency B afterward and buy another two tickets there as well, which result in a total of six tickets. When the agencies are connected again, the honest officers would recognize the harm - the constraint has been broken! Hence, distributed systems often have the attempt of not servicing when not all systems are connected correctly. This may lead to long breakdown times of the whole system. In such a case only a few soccer fans are not able to buy their tickets. The worst case is that they are not allowed to watch the game live in the stadium. But there exist other, more critical applications, like flight booking applications of airlines where longer downtimes could lead to a massive loss of money and negative press headlines.

Thus, the Dependable Distributed Systems project (DeDiSys) — a European research project — follows a different approach while systems are not fully connected but need to serve requests. In short, it allows the temporary breaking of constraints during network failures. Therefore, Ulrich is allowed to buy as much tickets as he wants in every agency he reaches during the network failure. But if the system gets back its full connection and resolves all the different purchasing of the diverse people, the constraints must hold their original intention again. In Ulrich's case, if he bought six tickets and did not make a storno, the system would cancel the last two purchasing so he would get only four tickets. The advantage is obvious: Even if systems act as islands not connected with the other systems, they still provide their services and store all the necessary information to get back a constraint satisfied system in case of full connection. Ideally, the system shall work only fully connected. However, in theory, theory and practice are the same, whereas in practice they are not. Hence, such approaches are needed.

If we rethink the story, this thesis implements a distinct part of such a special fault tolerant distributed system. The DeDiSys project is not a ticket application, that was just a hint for better understanding. The implementation that is part of this thesis stores the needed information for later constraint reconciliation during the time the systems are not fully connected. If the systems are joined, it provides a better performing protocol for exchange of the different data items as the prior existing solutions presents. It is realized that different rollback or compensation mechanisms are provided, which contribute to the finding of a constraint satisfying solution in case of constraint inconsistencies. Finally, these functionalities are evaluated and discussed.

## 1.1 Dependable Distributed Systems - DeDiSys

The DeDiSys project is an international research project with the aim of finding a trade-off between consistency and availability in dependable distributed systems. A part of this project is a practical implementation to proof the ideas and concepts of the DeDiSys project with the help of a prototype software package. This software consists of client applications (such as a flight booking application) and a framework which is hooked in a free Java application server (JBoss). The framework is built against the Java 2 Enterprise Edition (J2EE) architecture and provides a number of features to optimize dependability by handling failures of nodes and connections in a distributed system.

The mission of this project is the following: For safety or mission critical systems, in which availability is more important for dependability than strict data integrity, availability shall be enhanced by temporarily relaxing data integrity. Potential inconsistencies are accepted by constraint validation on replicated copies. These copies are potentially stale in the face of network partitions. A following reconciliation phase will eventually resolve these threats, see (Froihofer et al., 2006).

## 1.2 Background and motivation

Replication is the process of maintaining multiple copies of the same entity (object, data item). It is well known that fault tolerance for improved availability in case of node and link failures is provided through replication. Link failures are especially troublesome as they may lead to network partitions, where only computers in the same partition are able to communicate. Such situations have to be addressed explicitly in partitionable replication protocols.

The primary partition protocol, see (Ricciardi et al., 1993), allows a single partition (the primary partition) to continue with the "normal" operation, while other partitions are blocked or operate in read-only mode. Such an approach prevents replica conflicts as write operations are only allowed in the primary partition. To further increase availability, write access in other partitions would be desirable — at the price of replica inconsistency.

Within the DeDiSys project, a primary-per-partition protocol has been developed that allows for replica inconsistency, see (Beyer et al., 2006). These inconsistencies have to be reconciled after node/link failures are repaired. This reconciliation is a complex task and primarily depends on the history of states/operations and the number of objects, partitions and constraints. This master's thesis implements a modified replication protocol based on a “standalone” history service in which both replication and history service are configurable to allow for improved behavior (e.g. better performance during degraded mode and reconciliation phase).

The evaluation section of this thesis compares the new implementation to the existing solution. It is expected that the new implementations show the following characteristics:

- The improved replication protocol shall provide faster propagation of missed updates in contrast to the given DeDiSys replication protocol.
- The rollback (compensation) to a new entity state shall - while the system tries to reach the healthy mode again - deliver a correct solution. Until now the rollback was only rudimentary implemented without functionality tests.
- This rollback (compensation) mechanism shall behave with high-performance.

## 1.3 Structure of this thesis

The structure of this thesis is as follows:

**Introduction.** The introductory section gives background information to the understanding of the specific task “Implementation and Evaluation of Partitionable Replication Protocols in an Enterprise JavaBeans environment”. A description is delivered which illustrates the motivation for this thesis and embeds it into the problem space of dependable distributed system - the focus of the DeDiSys project.

**Technology Overview.** In the section “Technology overview“, a number of theoretical bases for technologies are described. Generally, an in-depth discussion will be avoided, but important facts for the comprehension of the following sections are presented.

**Research** The section “Research” describes the DeDiSys project, aspects of replication protocols and related work.

**Implementation.** The section “Implementation” details the different aspects of the solution and marks it off future implementations. It includes the new replication protocol, the history service, rollback and compensation phase and needed external applications.

**Evaluation and Results.** The used test environment and the evaluation tests with their description are presented in this section. Additionally, the test results are discussed in detail.

**Discussion.** “Discussion” provides arguments regarding the implementation phase and test results. It contains the author’s view about the different implemented modules under consideration of the aims of this thesis. Additionally, recommended future work is presented.

**Conclusion.** In the section “Conclusion” the whole work is summed up. Furthermore, the most important design issues and the conclusion can be found.

**Appendix.** In the “Appendix” one can find the link section.

# 2 Technology Overview

This section provides information about technologies which are used to implement the required functionality of this master's thesis. As this thesis is part of the DeDiSys project, it is necessary to mention that the project itself makes use of three different base technologies to show the feasibility of the approach: With the usage of EJB (Java), .NET (C#) and CORBA (Java), the trade-off "consistency versus availability" is investigated for a tightly-coupled data-centric distributed systems. With this project background this thesis focuses on EJB as its underlying technology.

During the last years the Java 2 Platform Enterprise Edition (J2EE) architecture became popular for distributed applications by providing services like transaction management, persistence and object life-cycles. The data items (objects) called entity beans are technology-specific and essentials in J2EE.

## 2.1 J2EE and EJB

The J2EE platform acts as an umbrella standard (no implementation!) for Java's enterprise computing. As illustrated in (Roman et al., 2005) it bundles technologies for a comprehensive enterprise-class service development and deployment in Java. Therefore, it is a robust suite of middleware services and is built up on the Java 2 Standard Edition (J2SE). A J2EE compliant implementation — as it can be found in an application server like JBoss — provides a number of services to the developers. The most relevant for this thesis are according to its usage within the implementation:

- Enterprise Java Beans (EJB): The EJB standard defines how components have to be written on the server-side. Additionally, it provides a contract between these components and the application server. EJB itself makes use of other J2EE technologies, as it constitutes a major part of the J2EE standard.

- Java Remote Method Invocation (RMI): Java's usual way of communication between distributed objects (objects on different computer machines) is handled via RMI. RMI-IIOP is a variant of RMI that communicates over the CORBA IIOP protocol.

- Java Naming and Directory Interface (JNDI): With this interface one is allowed to look up EJB components and other resources within an application via convenient names.

Generally, the EJB technology is built upon the two mentioned standards: Java RMI and JNDI, ee (Roman et al., 2005) for further details on JDNI and Java RMI.

But what is an enterprise bean? An enterprise bean is a server-side software component which is deployable in a distributed environment. It can be an arbitrary complex piece of software, consisting of more than one object. Independent from the bean's actual size, the client application communicates via a well-defined interface (which conforms to the EJB standard) with the bean. Hence, the real size and complexity of the enterprise bean is hidden. The client software can be anything such as a servlet, an applet or another bean.

Although the EJB standard comes with version 3.0 now (DeMichiel and Keith, 2006), the DeDiSys EJB implementation is based on the prior standard EJB 2.1 (DeMichiel, 2003).

There are three different types of beans defined:

- Session beans: These beans model business processes, performing actions on objects. The actions could range from database access to calling other beans.
- Entity beans: Entity beans model business data. They can be seen as data objects. Typically, entity beans are utilized by session beans to access and modify persistent data.
- Message driven beans: These beans can be compared to session beans, as they perform actions. This only happens in an implicit way by sending messages to the beans.

**Distributed objects.** The basis for EJB components are "distributed objects". These objects are callable from a remote system. Figure 1 shows the communication path from a remote client to an EJB component in an application server:

At first, the client calls the stub, which represents a client-side proxy object. This stub is liable for masking the network communication from the client. It has knowledge about how to communicate over the network. Now the skeleton (a server side proxy) is called by the stub over the network. This skeleton masks the network communication from the distributed object. Finally, the skeleton forwards the call to the distinct object representation. After the object has done its job, the call goes the other way round back to the client. It is important to note that both the stub and the object on the server implement a so called "remote interface". Therefore, the client assumes it calls the remote object directly, while in fact the stub implements the remote object interface, see (Roman et al., 2005) for further details. Theoretically, if a client never sees a difference between local and remote interaction, this could be called "distribution transparency". In practice (such as this thesis implementation) this is not true, as it is only a simplified approach. For example, message latency between distributed nodes is usual due to a higher network traffic or a link error . Thus, the client application is aware of a slower message transfer (or error handling in case of the link error) compared to a single-node system and transparency is uncovered.

**Enterprise Bean.** Each enterprise bean class implements the `javax.ejb.EnterpriseBean` interface. This forces the bean to expose a number of methods (e.g. `create()`) as defined in the EJB standard. An EJB container (this could be an application server) calls this method to manage the bean and its life cycle. According to the specialization, session

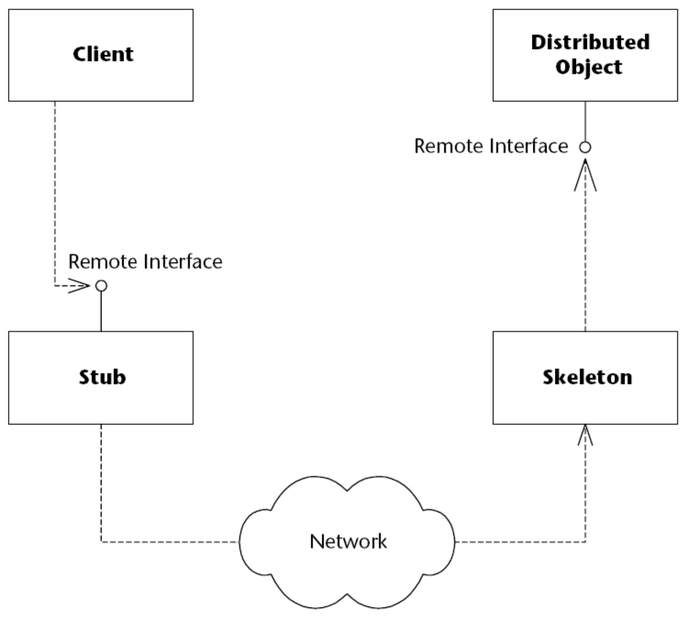


Figure 1: Distributed objects, (Roman et al., 2005)

beans, entity beans and message-driven beans have specific interfaces which make their behavior distinct within the container.

The communication with an enterprise bean follows a given path: The client never invokes an object directly. Instead the invocation is intercepted by the container which holds the bean. This allows middleware services to be performed implicitly, e.g. transaction management, persistence, threading, location transparency etc. Hence, the implementor of the business logic can concentrate on its task and does not need to program against the middleware API. To sum it up, the EJB container provides a level of indirection between the client code and the bean. This layer of indirection — the glue between the client and the bean class instance — is called the "EJB object" and acts as a request interceptor.

Due to the fact that the implementation makes use of session beans and entity beans, they are illustrated briefly:

### 2.1.1 Session Bean

A session bean represents work being performed for the client that is calling it. They are business objects that implement business logic, business rules, algorithms and so on. A key difference between a session bean and an entity bean is their life cycle: The instance of an session bean is a short-living one. It has approximately the same lifetime as the "session" - the time the client is communicating with the bean. The session beans are not shared between different instances. The application server (the container) maintains this life cycle.

There are two subtypes: stateful session beans and stateless session beans. Stateful session beans are designed to maintain business processes that include multiple method requests or transactions. Stateless session beans, on the other hand, hold conversations that span only one method call; so they do not hold multiple method conversations from the client, see (Roman et al., 2005) for further details.

Within this thesis Session Beans are used, for example, within the history service.

### 2.1.2 Entity Bean

Entity beans can be seen as persistent objects that might be placed in a permanent storage (persistent objects are able to render themselves into a persistent storage). They use persistence mechanisms like e.g. serialization. Generally, such objects represent "data" (simple or complex). Entity beans have — in contrast to the session beans — a client-visible state and their lifetime might be completely independent from the client application lifetime.

There exist two types of persistent management mechanisms for entity beans:

- Bean managed persistence (BMP): The entity bean itself (and the developer respectively) is responsible for persisting the data to an arbitrary storage.
- Container managed persistence (CMP): With CMP there is no need for the programmer to write a storing relevant code. The EJB container handles all required database calls. It is the favored persistence mechanism within the DeDiSys project.

Within the DeDiSys project, entity beans are used in multiple ways:

- Entity beans are used as data structures for objects from external applications, e.g. PassengersBean, InvoiceBean and FlightBean.
- Within the constraint consistency management entity beans are used for e.g. storing of threats or affected objects.

## 2.2 JBoss

JBoss, with its actual version 4.2.1, is an application server, see (JBoss-Inc., 2006). It is fully compliant to the J2EE architecture and consists of different parts: An underlying MBean server which manages components, the microkernel and service components such as MBeans (MBeans are monitorable resources; for each MBean, a management interface must be defined, those interfaces are used by the MBean server). Therefore, different configurations can be afforded and adapted to a given requirement. Components which are not needed can easily be removed or, if needed (again), plugged in. JBoss comes out of the box with three configurations:

- minimal: This represents the minimum services to start the JBoss server; it is intended for projects that do not need further J2EE implementation in addition to self-written services.

- default: The standard services that are needed by most J2EE applications get executed.
- all: All possible components are booted.

The DeDiSys framework makes use of a self-defined configuration named “dedisys”. It provides the implementation of concepts for concurrent access in different network partitions, based on fault-tolerant, partition-aware replication and explicit management of data integrity constraints. For using the “dedisys” configuration, a few steps are necessary in advance: The group communication toolkit “Spread” (see appendix 7) must be downloaded and installed correctly. The files compmonitor-config.xml and history.xml have to be adapted. Additionally, the database must be configured.

**JBoss interceptor architecture.** JBoss places an interception mechanism between client and server. An interceptor is represented through an event-triggered Java object. Multiple interceptors form an interceptor chain with a defined order. Each interceptor in the chain acts like a filter which modifies, audits or just forwards a request. Interceptors make use of services like transaction, security or persistence. If an interceptor object is triggered by an invocation object, a defined operation is performed on that invocation object. Afterward, it is passed to the next element in the interceptor chain. The advantages of adding filters to a chain are configure ability and ease of use: First, new interceptors in JBoss have to implement the org.jboss.ejb.Interceptor interface. Second, the interceptor chain with a defined order is changed in the file standardjboss.xml in the JBoss configurations. If a new interceptor is needed, an alteration of the XML-file adds the needed behavior to the framework without changing the code in other parts of the system.

## 2.3 Java Management Extensions - JMX

The JBoss server is completely built upon component based plug-ins, see (JBoss-Inc., 2006). This modularization is supported by the use of Java Management Extensions (JMX). JMX is the standard for monitoring and administrating different Java components. Accordingly, the different MBeans components may be administered using JMX. The advantage of this approach is the high composability a developer has while integrating its solution for a specific problem: Components are plugged in if needed, otherwise they are removed to provide a small memory footprint.

### 2.3.1 Principles of JMX

The JBoss JMX implementation identifies the JMX integration as the integration spine which plugs components together. These components are MBeans that are loaded into the JBoss.

**Three levels of JMX model.** There exist three levels which describe the JMX model:

- Instrumentation level: This level defines the demands for JMX manageable resources like applications, service components or devices. Resources can be handled with the help of a Java object or wrapper that points out the manageable features to instrument it by a JMX compliant application. A notification mechanism is also specified in the instrumentation level. This allows MBeans to correspond with their environment about e.g. state change notifications.
- Agent level: Agents in this context are responsible for controlling the administered resources, which are registered by the agent.
- Distributed services level: This is a mechanism which defines the administration of applications with agents and the objects they manage.

### 2.3.2 JMX components

JMX components can be found in the instrumentation level (MBeans, notification model elements and MBean metadata classes) and the agent level (MBean server and agent services).

**MBeans.** MBeans are Java objects that implement a standard MBean interface and comply with predefined design patterns. An MBean for a resource adds the necessary information and operations that an administration application needs in order to manage this resource. JMX defines four types of MBeans to support a number of instrumentation needs: Standard MBeans, Dynamic MBeans, Open MBeans and Model MBeans. For the DeDiSys project only standard MBeans are used. They use statically defined management interfaces and a simple naming convention. As described above: With the help of MBeans one can extend JBoss functionality with self-written custom services.

**Notification model.** JMX notifications are an enhancement of the Java event model. The MBean server and MBeans can communicate via notifications to provide information. In the specification the definition of operations on the MBean server are included. These operations are needed for the registration of notification listeners.

**MBean server.** An MBean server is a registry unit for MBeans. The interfaces of given MBeans are made available for use. Loose coupling between management applications and the MBeans is provided due to the fact, that MBeans do not expose the MBean object. Rather the interface (metadata, operations) is disposable via the MBean server interface.

**JMX in DeDiSys.** The DeDiSys project makes use of the MBean technology in its constraint consistency manager service (CCMgrSvc, see (Horehled, 2006) for further details). Additionally, parts of the history service are implemented with the help of JMX, see Section 4.1.3 for further details.

## 2.4 ADAPT

As declared in (Shannon, 2003), the component model (e.g. session and entity beans) of J2EE supports the development of replication integration. Application states and clearly declared objects form an inseparable combination. The application server handles the invocations to this objects and manages the transactions in a centralized way. As the object communication is exposed to the server, it may present all this information to external replication algorithms. As a matter of fact, design and evaluation need considerable effort in development. Additionally, integration of such algorithms into an existing application server requires much internal knowledge. Any modification to the server must be rechecked when a new server version arises. Not to forget that replication protocols require a bunch of work which is common to all of them — such as the interception of component invocations from outside the server.

Hence, the ADAPT project (Babaoglu et al., 2004) developed a two-layer strategy. The handling (lower layer) with the underlying application server is transparent for the replication algorithm (top layer). Therefore, different pluggable algorithms may use one layer which is specifically implemented for a server architecture. Important is the fact that the J2EE implementation is not visible for the top layer. When an invocation occurs to a given component, the ADAPT framework passes control to the replication algorithm, before it reaches the component.

In short, there are two advantages for the decoupling of replication algorithm and J2EE application server:

- ADAPT provides building blocks for a number of replication algorithms (e.g. the interception of calls to components).
- It hides specifics of the given application server.

**DeDiSys and ADAPT.** The integration of ADAPT into the DeDiSys middleware is a logical step due to the following reasons:

- The notification of replication-relevant invocation events between a client invocation to an object and an EJB Entity Bean implementation through the JBoss has a new ClientComponentMonitor (Client to JBoss) and a ComponentMonitor (JBoss to Entity Bean) interface integrated. These interfaces are needed for the replication protocol to get notified about the distinct invocations. Additionally, ADAPT provides a ComponentHandle which allows to perform some generic methods on the components of interest, see (Künig (ed.), 2007) for further details.
- ADAPT allows the usage of Spread as an underlying group communication tool.
- The DeDiSys replication service is directly built upon the ADAPT interfaces.

# 3 Research

This section provides information about the DeDiSys project, replication protocols and related work.

## 3.1 The DeDiSys middleware approach

This subsection describes necessary items of the DeDiSys middleware approach (regarding this thesis). As mentioned, there are three different technologies used in the DeDiSys project. Common for all three technologies is the fundamental theoretical system modeling. The three technologies differ mainly in the way the model components are implemented (regarding the technology specific conditions), used protocols in the reconciliation phase and generally the complexity and depth of the implementation. (Froihofer (ed.), 2007) shows that the EJB approach is the most comprehensive one, regarding the implemented functionalities and mechanisms. Within a previous evaluation, it has been shown that a rework of the reconciliation phase is desirable to show a possible rollback and/or compensation strategy of replica and constraint inconsistencies.

### 3.1.1 Trade-off between consistency and availability

The focus of the DeDiSys project is the investigation of the trade-off between consistency and availability. Additionally, this trade-off shall be configurable. The idea behind this is to reduce the consistency temporarily, thereby improving availability. Most of the work done in this area comes from (mobile) databases systems. DeDiSys investigates distributed object systems with nested invocations, see (Froihofer (ed.), 2005a). There are two extremes regarding this topic - the optimistic and the pessimistic approach, see (Davidson et al., 1985):

- Pessimistic strategies prevent inconsistencies by limiting availability. Each partition makes worst-case assumptions about what other partitions are doing and operates under the pessimistic assumption that if it is possible that an inconsistency occurs, it will occur.

- Optimistic strategies do not limit availability. Any transaction may be executed in any partition that contains copies of the items. Within the DeDiSys project this is the favored principle, see (Osrael et al., 2006). Optimistic strategies differ mainly in how they detect and resolve inconsistencies.

In a partitionable system, nodes may become separated, because of a link failure. Through replication of objects such a system allows operations on an object even if the host with the primary copy of the replica object is in another unreachable partition. Thanks to replication, the system can act like it were fully connected. It is worth mentioning that the replica is not just a 1:1 copy of the entity on each node, but a placeholder which holds the actual state of the object in the distinct partition. It is a matter of fact that objects in different partitions may differ while time passes (this is dependable on the replication protocol, but true with the protocol implemented and enhanced within this thesis). Additionally, different objects might require different consistencies. Therefore it is possible that some operations are not allowed on objects during a split system. This limits the availability of the whole system.

The DeDiSys project regards three different consistency types (Froihofer et al., 2007):

- Constraint consistency: It controls data integrity via a number of integrity constraints. The system is fully constraint-consistent, if only changes to objects caused by operations whose constraints have been met are accepted.
- Concurrency consistency (isolation): This belongs to the ordering of concurrent operations on objects to avoid conflicts.
- Replica consistency: This refers to the extent to which replicas of a single object differ. A system is fully replica-consistent, when all replicas of each individual object have stored the same state. This depends on the used replication protocol. The ones used within this thesis follow that approach.

The DeDiSys approach focuses on constraint consistency and replica consistency and allows them to be relaxed while partitions occur. Concurrency consistency is in a lower layer and not handled directly by the middleware, see Figure 2.

### 3.1.2 DeDiSys system model

The DeDiSys system focuses on the data-centric approach for distributed systems. Relevant details that help to understand the design approach and concepts that are used in this master's thesis will be described in the following, see (Froihofer (ed.), 2005a).

**System states.** Generally, there exist three different system states: After its startup, a system is in the "degraded" mode, as long as not all other defined nodes are connected. A partition is a number of nodes which are connected together with some nodes missing to reach the intended working mode - the "healthy" mode. During the "healthy" mode the system is fully consistent. This may change, if nodes disappear (e.g. through link failures) this way leading again to the "degraded" mode with a number of partitions. During the "degraded" mode, inconsistencies are allowed up to a certain degree, as long as they are controlled by integrity constraints. After reconnection, the system is in the "reconciliation" mode, in which first replica consistency and then constraint consistency shall be reached. This, in turn leads to the "healthy" mode again, see Figure 3.

**Consistency and constraints.** The constraint classification within the DeDiSys project are described in (Froihofer et al., 2007): Preconditions have to be validated before the call to a method, postconditions have to be satisfied after the call to a method returns. Invariant constraints are defined solely on the state of objects (static constraints) and

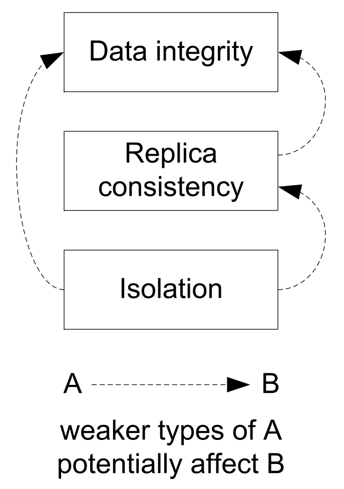


Figure 2: Consistency types, (Froihofer (ed.), 2005b)

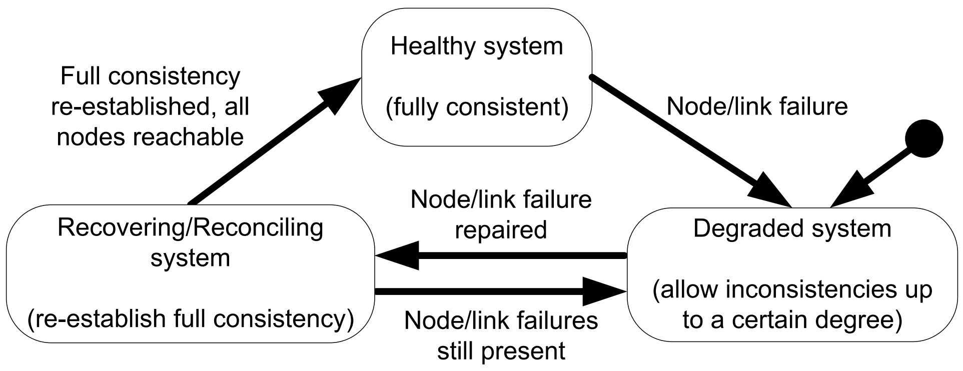

Healthy system
(fully consistent)
Full consistency re-established, all nodes reachable
Node/link failure
Recovering/Reconciling system
(re-establish full consistency)
Node/link failure repaired
Node/link failures still present
Degraded system
(allow inconsistencies up to a certain degree)

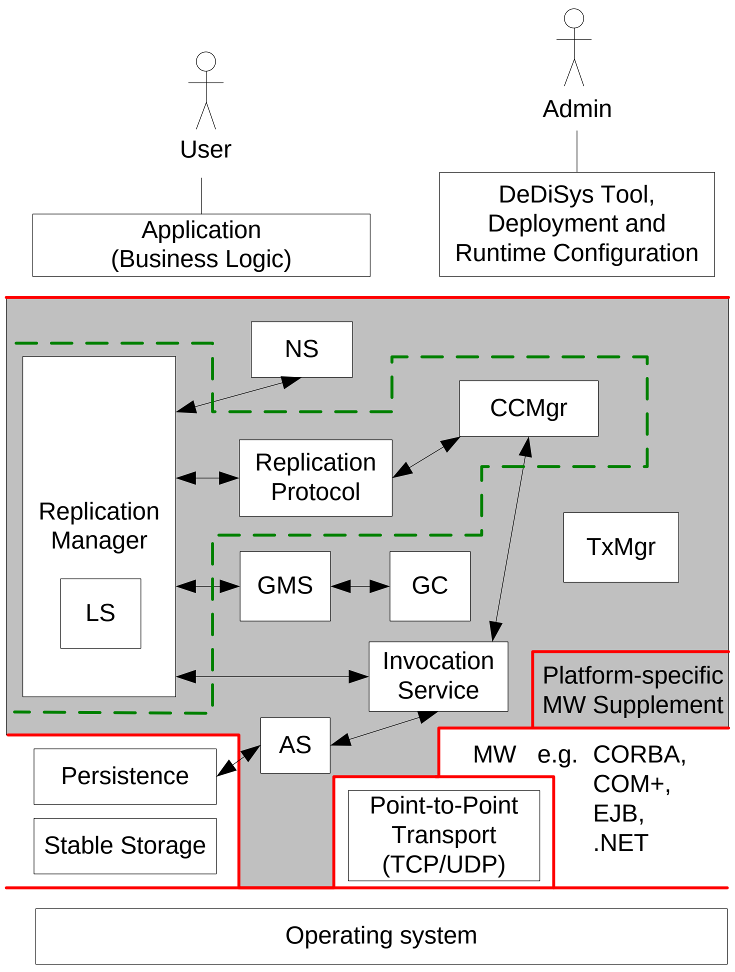
Figure 3: System states, (Froihofer et al., 2007)

hence can be validated at any time. Dynamic constraints defined on state transitions, sequences or temporal predicates are not in the primary focus of the work. An invariant constraint must be checked immediately after the call to a method which might change the state constrained by this specific invariant — an affected constraint of the method. Vice versa, the method is an affected method of the constraint. All objects restricted by a certain constraint are affected objects of the constraint. The validation of a constraint requires access to all affected objects. Additionally, the constraints are split in tradeable and non-tradeable constraints. Non-tradeable constraints are critical for correct operation of the system and must never be violated. Tradeable constraints must be satisfied in a healthy system — during degraded mode, however, they might temporarily be relaxed in order to increase availability. The decision of whether a constraint is tradeable has to be provided by the application developer according to an application's requirements.The reliable evaluation of a constraint requires an up-to-date version of all objects that participate in the constraint. During partitioning, secondary copies (copies of other nodes than the primary replica lives in) of a given object are out of date, if the primary copy resides in another partition. Missed update propagation in the reconciliation mode of the system may belated violate a constraint. Hence, a number of operations accomplished during partitioning may be undone to restore system consistency. This behavior is acceptable for most of the operations, but a small number of operations requires another characteristic. These "critical operations" include operations on data items which require consistency during the whole system lifetime plus operations which cannot be undone (e.g. because of side effects). Critical constraints handle such situations. These constraints must not be evaluated based upon possibly stale objects. An object is possibly stale, if write access is possible in another partition. Additionally, during the conflict resolution phase (reconciliation), critical constraints must not be violated, see (Froihofer et al., 2007).

**Architectural components.**

Within the system, a number of components are necessary to form the whole framework. Elements relevant for this thesis are the following, see (Osrael et al., 2006):

- Group membership service: This service knows which system nodes are reachable by a given node. The view for connected members is the same. It is essential to be informed about the connectivity status during the different system modes.
- Group communication: Multicast (or probably unicast) is also provided via the group communication module. Reliability is a very important aspect in this respect. As its underlying group communication toolkit the EJB approach in the DeDiSys middleware uses "Spread". This is an open source toolkit that provides a high performance messaging service.
- Replication manager: This module provides functionality, which is used by the different possible replication protocols, like the adding or removing of replicas. It


User
Admin
Application (Business Logic)
DeDiSys Tool, Deployment and Runtime Configuration
NS
CCMgr
Replication Protocol
Replication Manager
LS
TxMgr
GMS
GC
Invocation Service
Platform-specific MW Supplement
AS
Persistence
Stable Storage
MW e.g. CORBA, COM+, EJB, .NET
Point-to-Point Transport (TCP/UDP)
Operating system

| | |
|---|---|
| AS | Activation Service |
| CCMgr | Constraint Consistency Manager |
| GC | Group Communication |
| GMS | Group Membership Service |
| LS | Location Service |
| NS | Naming Service |
| MW | Middleware |
| TxMgr | Transaction Manager |

DeDiSys ▨

Figure 4: Basic architectural components, (Osrael et al., 2006)

is connected with many other modules, such as the naming service, the replication protocol, the group membership service and the invocation service.

- Replication protocol: Its main tasks are to manage update propagation between systems as well as the timing of these updates. Together, the replication manager and the replication protocol form the replication service; see (Froihofer (ed.), 2005a).

- Constraint consistency manager (CCMgr): The idea for constraint consistency management was newly introduced in the DeDiSys approach. Together with the replication service it makes up the core functionality of DeDiSys. It validates the constraints, triggers the negotiation of consistency threats, supports application specific constraint reconciliation and stores information about constraint consistency threats. Constraint consistency management is a completely new approach in this form, therefore no existing frameworks could be used. However, as there are no restrictions through underlying frameworks, the developer of the CCMgr has only the limitations of the EJB environment.

- Invocation module: The invocation service allows the interception of object invocations. It is provided with the JBoss core functionality and can basically be seen as a filtering and transformation principle within object communication. This is especially useful for the history service implemented within this thesis. The handling of the operation history is achieved among this. However, its main task is the invocation logic used for the invocation of methods. As JBoss already includes an invocation service with the option to register interceptors server wide and in client applications, it is not necessary to implement a new one.

- Persistence: Generally, persistence provides fault tolerance in case of node failures. Within the EJB implementation, bean objects, replicas and history entries are not only held in memory, but are stored in a database. The framework makes use of an underlying database, for which MySQL and HSQLDB are used.

Figure 4 illustrates the interaction of these components.

## 3.2 Replication and Replication Protocols

According to (Osrael et al., 2006), a distributed system supported by the DeDiSys framework consists of a number of nodes, i.e. computers that are addressable via an IP-address. One example application implemented on top of DeDiSys is a flight booking application. This application allows the adding and modifying of flights and bookings with parameters like “passenger”, “credit card number” and so on. During normal operation, all nodes are connected together and the system works in the so-called “healthy” mode. That means that if one flight booking client on a node adds a replica (or modifies it), all other nodes get the information and update their backup replica. Hence, all nodes share the same information. If one or more nodes are disconnected — for example due to network failures — the system mode changes to “degraded”. Nodes which modify replicas in the “degraded” mode are informed about possible threats. If two nodes in different network partitions change values of the same person, two inconsistent replicas are the result. These replica inconsistencies have to be resolved after the nodes are connected again. During this reconciliation process, the replication protocol executes its “missed update propagation” and discovers the conflicting situations. It calls a replica reconciliation handler that solves the conflict by inspecting the replicas. Different strategies are described in (Ertl and Ji, 2006). When all replica conflicts are resolved, the system state is “healthy” again.

**Distributed systems replication.** There exist active and passive replication. (Wiesmann et al., 2000) provides a description:

- Active replication: This is a non-centralized replication technique. All replicas receive and process the same sequence of client requests. Server process requests in a deterministic way, which guarantees consistency. Determinism is reached when the same operations are applied in the same order and all replicas will produce the same result. The main advantages are its simplicity and its failure transparency. The main disadvantage is the determinism constraint, as it is hard to achieve.

- Passive replication: It is also called “Primary Backup” replication. Clients send their requests to the primary node, which executes the requests and sends update messages to the backup nodes. The backups do not execute the invocation, but apply the changes produced by the invocation execution at the primary (e.g. updates). Thus, no deterministim is necessary for the invocation execution. Passive replication (and active replication, too) allows multiple threads, as the communication between the primary and the backup nodes has to guarantee that updates are received and processed in the same order. In the DeDiSys approach passive replication is the favored scheme.

To facilitate the understanding of replication protocols, a few definitions are needed: (Osrael et al., 2006) and (Froihofer (ed.), 2005a) describe the primary mode of the replication protocol.

**Replication protocols.** Replication protocols are specified by three different activities:

- Update propagation: Update propagation describes how the updates on the worker object are propagated to the backup objects.

- Preparation for reconciliation: Missed updates for replicas that have failed or are in different partitions are stored.

- During reconciliation: It has to identify replicas that missed updates and apply the updates to them, trigger the re-evaluation of constraints after the updates are applied and handle write-write conflicts.

(Froihofer (ed.), 2005b) defines replication and reunification as follows:

**Replication.** Replication is the process of maintaining multiple copies of the same entity (data object) at different sites (=nodes).

**Reunification.** Reunification is the process of merging two or more partitions on the network layer, i.e. communication between the partitions is re-established.

**Reconciliation.** Reconciliation is the process of detecting and solving inconsistencies and conflicts caused by updates of replicas of different partitions. This is necessary after reunification. According to (Froihofer (ed.), 2007), reconciliation is one of the difficult problems systems have to solve with optimistic replication.

This thesis uses these definitions as provided above.

### 3.2.1 Functional model of an abstract replication protocol

In (Wiesmann et al., 2000) a functional model with five generic phases is illustrated: Mapped to a specific protocol, the implementation of some phases might be skipped or the order appears in a different manner. So the protocols can be compared by the way they implement each of the phases and the way they combine the different phases. Figure 5 shows the complete process.

- Request: A client submits an operation to the system. There exists a clear distinction between replication strategies which send the operation directly to all backup copies (active replication) or to only one copy (passive replication). If passive replication is chosen, the replica that received the message is responsible for forwarding it to all other replicas as part of the server coordination phase.
- Server coordination: The different replicas try to find an order in which the operations are to be performed. This is one of the most diverging points in replication protocols, as it depends on strategies and mechanisms of order and correctness criteria.
- Execution: The execution phase represents the actual performing of the operation.
- Agreement coordination: It is the task of this phase to guarantee that all different replicas ensure they all do the same thing.
- Response: The client receives a response from the system. In distributed systems the response usually takes place after the protocol has been executed so that no discrepancies arise.

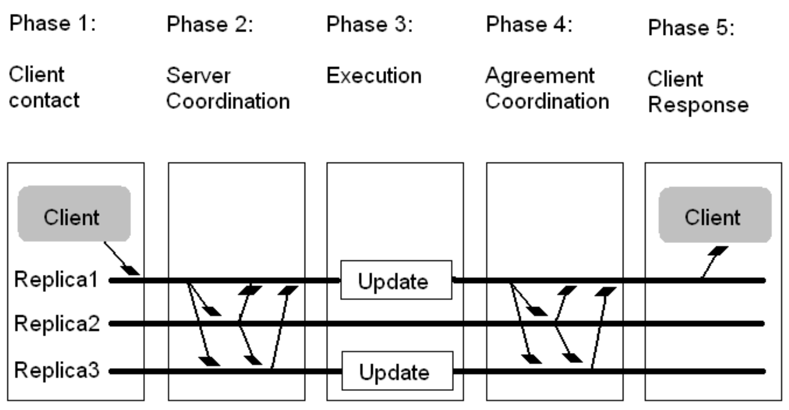


Figure 5: Five phases of an abstract replication protocol, (Wiesmann et al., 2000)

### 3.2.2 Primary-per-partition replication protocol

During the DeDiSys project, a primary-per-partition replication protocol (Beyer et al., 2005) has been developed. Unlike the primary-partition protocol, the primary-per-partition protocol allows consistency (based on integrity constraints) to be temporarily relaxed. This increases the availability in a replicated distributed system. If the system is partitioned, operations in each partition are allowed. This approach has two advantages: First, the availability of the system is increased for applications. Second, operations can proceed independent of the given partition of the object. This protocol is a relaxed passive replication protocol. That means that read-only operations are allowed on any replica, whereas write operations are directed to the primary copy of the object being accessed. If the primary copy of the object is in a different partition, a secondary copy is promoted to a temporary primary.

**Normal mode.** During the system's normal mode all object write invocations have to be directed to the primary replica. The different object invocations are combined to a transaction. The pre-condition constraints associated with the operation are now evaluated. By assuming a constraint is not met, the whole transaction is aborted. In the normal mode temporary relaxing of constraints is not allowed. After the primary replica has updated its local state, the post-conditions and invariant constraints of this operation are evaluated. Similar to the pre-condition: If a constraint is not met, the whole transaction is aborted. If the operation succeeds, all primary updated replicas propagate their object states to the backup replicas.

**Degraded mode.** In case of a node or link failure, the system may be split in a number of partitions. Write actions during the degraded mode are handled similar to the normal mode, but with a few enhancements: If the primary copy of an object being written cannot be found, a secondary copy is chosen (called "temporary primary"). Transactions with critical constraints are aborted immediately. Regular and relaxable invariant constraints with possibly stale objects get a marker for re-evaluation during the reconciliation phase.

**Reconciliation theory within the primary-per-partition replication protocol.** This concerns the theoretical approach of the DeDiSys reconciliation phase. As it forms an essential part of this thesis, the different implementation descriptions of the three frameworks EJB, .NET and CORBA are provided afterward. According to (Froihofer (ed.), 2007), their difference in the protocol is not caused by the different platforms, but rather by different application scenarios implemented on top of the middleware systems. Only the EJB implementation gets enhanced during this thesis.

If at least two partitions re-join the reconciliation phase is started.

- At first, the replica consistency is restored. A write-write conflict occurs, if at least two primary copies of one object have been changed in different partitions. This conflict can be solved with a so-called reconciliation handler. It was part of former laboratory work to implement a reconciliation handler with a graphical user interface to choose either a given replica or create a new one, see (Ertl and Ji, 2006) for further details. Other reconciliation handlers are variants: The primary replica always wins, or the higher version replica wins.
- When the replica consistency is reached (now all replicas of one given object in the re-joined partition are identical), the constraint consistency is the next step. The markers for the constraints, which are created during the degraded mode, are re-evaluated, if the primary copy is now available. In case of a constraint violation, the application is asked for conflict resolution.
- Finally, the secondary copies are updated according to the changes of the primary copies of the data items.

**Reconciliation in EJB.** As described above, the EJB implementation forms the basis for this master's thesis. It shall provide the automatic choosing of replica states plus the manual fixing of conflicts. This has been partially implemented in prior work (Fuchshofer, 2006), but required reconsideration and adaption which was part of the research conducted for this thesis. Additionally, automatic operation-based compensation is provided. The protocol within the EJB environment is a two-step protocol. At first, replica consistency is established, afterward constraint consistency is reconciled. The implementation differs slightly from the theoretical approach. This concerns, for example, the propagation of missed updates to restore replica consistency to reduce the implementation effort. Instead of comparing the version vectors, the missed updates are propagated immediately from one partition to the other. There exist two automatic strategies for the replica choosing during replica reconciliation: The primary version wins or higher version wins (the version that is updated more often). This is a configurable parameter and depends on the application. The implementation of new reconciliation handlers (a new application with new demands may arise) is a more or less straightforward task with well-defined interfaces. Hence, other policies are possible. The next step is the constraint reconciliation to reach a constraint consistent system state. A generic rollback of objects to previous states has been implemented rudimentarily. One aim of this thesis is to complete and evaluate this approach. Concerning availability it can be said that new operations on a data object are allowed to perform directly after the replica states have been chosen. However, the system stays in an inconsistent state as long as the constraint consistency has not been reached. Therefore, operations performed on objects might be rejected, if the system is in replica consistent state, but constraint reconciliation has not been finished.

**Reconciliation in .NET.** According to (Froihofer (ed.), 2007), the .NET platform of DeDiSys has implemented the automatic rollback and the automatic state merge. Automatic rollback was used together with a synthetic application and a quorum-based replication protocol. This so-called “Adaptive Voting” makes use of the traditional voting scheme, but allows non-critical operations during degraded situations, even if the quorum cannot be fulfilled. This allows replica inconsistencies and data integrity violations during the degraded mode. The reconciliation phase is a two-step strategy: First, replica consistency is reached via the higher-version-win algorithm (the partition with more updates in a write-write conflict). The second part of the policy checks if data integrity is fulfilled. If not, a stepwise rollback to previous versions is performed until the given constraints are satisfied. In (Osrael et al., 2007b) Adaptive Voting is presented in detail. The automatic state merge reconciliation protocol can merge the replica states without having to chose one replica over the other. There are no integrity constraint violations that need to be solved either.

**Reconciliation in CORBA.** As illustrated in (Froihofer (ed.), 2007), the CORBA platform provides automatic operation based reconciliation and the automatic choosing of replica states plus the manual fixing of conflicts. The P4 implementation that performs manual reconciliation mechanisms is comparable to the EJB implementation. At first, replicas are chosen from each partition to establish replica consistency. Then all possibly violated constraints are re-evaluated. Application callback is used to resolve the violated constraints. In addition to this approach an operation-based reconciliation protocol is implemented. This so-called “Continuous Service” (CS) protocol is a variant of the P4 protocol, which logs all operations that are performed during the degraded mode. After network reunification the CS protocol sets up a sandbox-environment. In this sandbox all application objects are integrated. Their state is set to the state before the network was split. Now all operations are replayed in the sandbox environment until a final state is reached. Important is the fact that the CS protocol holds up service during the reconciliation phase. Reconciliation is performed in the background while the appearance of a network partition is simulated during this process.

## 3.3 Related Work

Similar to the DeDiSys project seeking for new insights within the research area "replication protocols" and "consistency versus availability", other interesting approaches regarding this subject were adopted in the past. A number of them will be presented here. The criterion for selection is their thematic relation to the topic.

### 3.3.1 Replication in Databases and Distributed Systems

Replication is an area of interest not only for distributed systems but also for databases. In (Wiesmann et al., 2000) an abstract and a neutral framework is presented to compare replication techniques from both communities:

Whereas replication in distributed systems is mainly studied for fault tolerance purposes, it is used in databases for performance reasons. The used mechanisms are similar. In the paper a model is presented; it allows to compare and distinguish existing replication protocols and distributed systems. Based on an abstract replication protocol, a variety of protocols is shown with their similarities and differences. Regarding the results, the protocols were parameterized and an accurate view of the problems is provided. Finally, this should lead to a good baseline for the development of new protocols.

Regarding this thesis the paper provides a theoretical basis for the abstract build-up of replication protocols. Whereas the used strategies are not convertible one-to-one to the needs of DeDiSys, it adds to a better understanding of the topic. An enhanced version of this model incorporating this trade-off is presented in (Osrael et al., 2007a).

### 3.3.2 Consistency in Partitioned Networks

The paper "Consistency in Partitioned Networks" (Davidson et al., 1985) surveys several strategies for transaction processing in partitioned distributed database systems with replicated data. They are compared under the competing goals of maintaining correctness and achieving high availability. At first, it is described how correctness adheres to availability: When a system that should operate when it is partitioned is designed, the competing goals of availability (the system's normal functionality should be disrupted as little as possible) and correctness (data must be correct when recovery is complete) must be reconsidered. As a next step, the notion of correctness in a database and database environment is described. The focus lies on transaction, and transaction handling. Optimistic replication protocols (e.g. which make use of on precedence graph to detect inconsistencies) and pessimistic replication protocols (e.g. via a read quorum voting) are shown. Finally, the guidelines for selecting a partition strategy sharpen the understanding of the issues within partitioned networks. Regarding the work load, (Davidson et al., 1985) conclude that optimistic policies work better when transactions are short and variance is small.

This paper provides many strategies for database replication (syntactic ones, one-copy serializability being the sole correctness criterion; and semantic ones, the semantic of the transactions or of the database defining correctness) for database replication. However, the DeDiSys approach "trade-off for availability and consistency" addresses how to solve the trade-off for for distributed systems and not for databases in an application-specific way.

### 3.3.3 Design and Evaluation of a Conit-Based Continous Consistency Model for Replicated Services

In (Yu and Vahdat, 2002) the semantic space between traditional strong and optimistic consistency models for replicated services is explored. It is argued that an important class of applications tolerates relaxed consistency, but benefits from bounding the maximum rate of inconsistent access in an application-specific manner. Thus, a conit-based continuous consistency model has been developed to capture the consistency spectrum via three application independent metrics (numerical error, order error and staleness). Additionally, the design and implementation of a middleware layer that enforces arbitrary consistency bounds among replicas using these metrics is presented. As mentioned in (Beyer et al., 2006), the application programmer is required to specify the required consistency or the required availability.

Although the conit-based approach for replicated services is an interesting one, it is not usable for the DeDiSys approach as it does not support partitions or integrity constraints.

### 3.3.4 Post-Partition Reconciliation Protocols for Maintaining Consistency

In (Asplund and Nadjm-Tehrani, 2006) a design exploration for protocols that are employed in systems with availability-consistency trade-offs is addressed. Central to distributed services in a data-centric application is how to maintain data consistency. Generally, the performance of such systems is affected by the replication and reconciliation protocols. The novelty approach here is a formal treatment of the reconciliation policy for achieving integrity consistency after network partitions. A formal description of three algorithms for reconciliation is provided. One algorithm is based on state transfer ("Choose states"), the other two algorithms ("Merging operation sequences" and "greatest expected utility") use operation transfer. Additionally, the algorithms have been evaluated by a theoretical study via metrics and through experimental studies. It is shown that the choice of reconciliation algorithm should be affected by parameters such as how long the average time is for the system to repair from network partitions. According to the authors, the number of nodes and objects is not relevant as long as it does not affect the complexity of the constraint. So one needs to take into account the impact on size on complexity of constraints.

This paper is an alternative research approach within the DeDiSys project. Its focus lies on the metric-based replay of operations during the reconciliation phase.

### 3.3.5 Measuring Availability in Optimistic Partition-tolerant Systems with Data Constraints

The paper (Asplund and Nadjm-Tehrani, 2007) describes implemented replication and automatic reconciliation protocols that can be used as building blocks in a partition-tolerant middleware. This approach allows the continuous service of the application during the reconciliation process. It is shown that the presented optimistic solution provides benefits in terms of availability in system with data constraints that have to be reconciled later on. Additionally, a prototype system is experimentally evaluated to illustrate the increased availability despite network partitions. The presented protocol gives the best performance for long partition duration. The costs for this approach are the revoking or compensation of operations during reconciliation.

This paper is an alternative research approach within the DeDiSys project.

# 4 Design Approach

This section presents the design approach of the implementation regarding this thesis' main focus. Generally, the programming and style of the design of the different modules is an object-oriented one — as the used programming language is Java. Whenever design patterns for the object-oriented approach are used, an accordant description is enclosed.

For a consistent composition, the sub-sections are generally structured in the following way with variances on demand:

- Motivation: A description why this software module is in the focus of interest is given.
- Requirements: The original requirements of the modules or those adapted during the design phase are presented.
- Functional Design: An in-depth technical discussion of the implementation is given.

At first, the history service is described. It is the basis for the mechanisms which shall provide a constraint consistent state. As the replication protocol for the replica reconciliation has been enhanced, the new functionality is illustrated. Afterward, operation compensation and state rollback are shown. These are processes which help to reach a constraint consistent system state. They are executed in the constraint reconciliation phase, which follows the replica reconciliation phase. A tool for setting replica states manually is the history application. It might be used after state rollback. The flight booking application is a prototype application where constraints are defined on and it has been adapted for the needs of this work. Additionally, the DeDiSys control application is illustrated. At last, the used system environment is described.

## 4.1 History Service

As the history service is the basis for the following software modules which were implemented, it will be introduced first.

### 4.1.1 Motivation for the History Service

The main target of the DeDiSys project is to allow write access on different — even non-primary replicas — in degraded mode, while node or link failures are present. If the system has a number of inconsistencies after a link or node error it has to solve these inconsistencies during the so called "reconciliation" mode. To fulfill this task, a service is needed which administrates the states and operations of replicas — the history service. This is reached with the logging and saving of changes of the given replica in a database. It constitutes a main task of this thesis to present the functionality in three different components in the DeDiSys project which partially fulfill this task: The replica object itself, the replica protocol and the constraint consistency management were used for this task in prior versions. It is necessary to centralize that kind of business to get a consistent and configurable solution for the whole DeDiSys EJB platform.

In Figure 6 one can see the horizontal distribution of the history service. Within a system it is directly used by the replication manager and the reconciliation module. An external application might use it via a session bean as an alternative access mechanism. As it is necessary to communicate over system borders, intermediate replication managers provide the possibility to get information of external history services. In contrast, external applications are able to use a distinct history session bean directly.

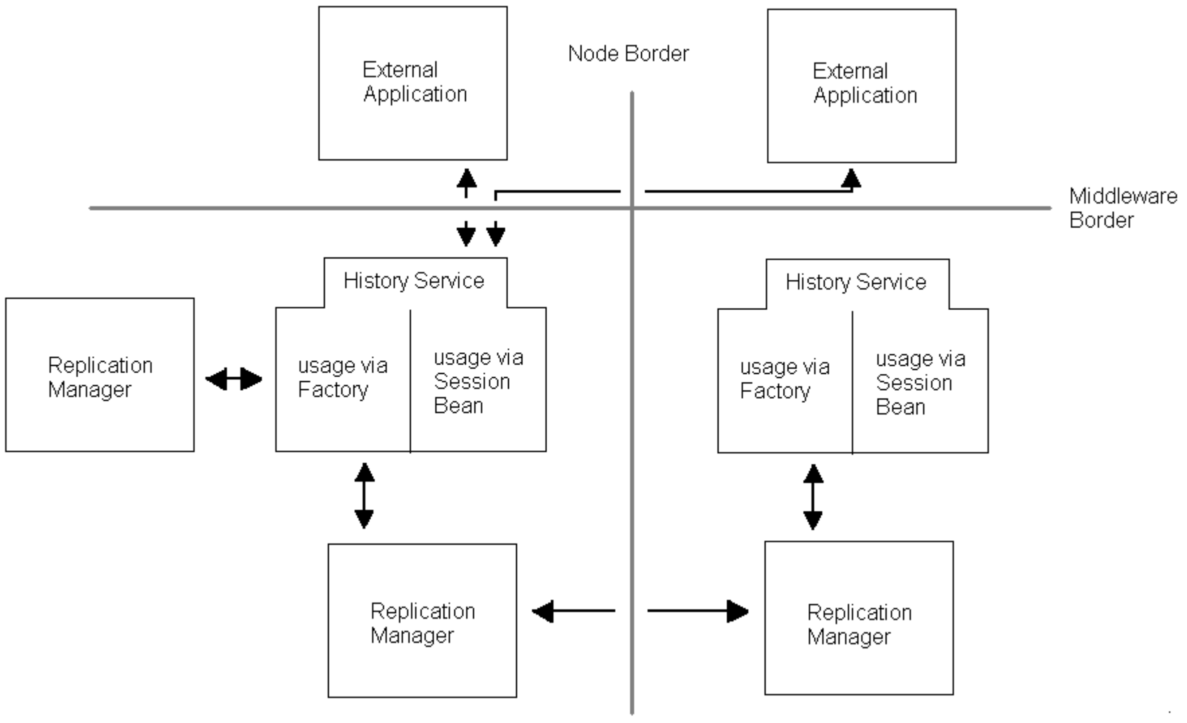


Figure 6: Horizontal distribution of the history service

### 4.1.2 Requirements for the History Service

The requirements regarding the history services in the DeDiSys EJB environment are the following:

**Stand-alone service.** The history service shall be a stand-alone service within the DeDiSys EJB middleware. As described above, different modules in the framework have implemented their specific “history service”.

**History Service interface.** The history service shall provide an interface for external applications and the DeDiSys middleware itself. This interface shall guarantee the access to the core history functionality, which creates snapshots of a given replica on demand or stores history entries.

**Configuration.** Configuration of the history service shall be possible before the nodes are started. All nodes shall have the same history configuration to guarantee a correct behavior during the running of the system. An “on-the-fly” configuration while the system is running is not useful: E.g. considering what happens, if suddenly no more history is saved and the reconciliation phase requires history entries for the rollback mechanism.

**The “degraded” mode for operations.** The functionality “history service in degraded mode for operations” shall be the correct working of a newly to implemented history service in the degraded mode with with the different modes “full” or “no” history for operation compensation. If “no history” is chosen, no operations shall be saved during the degraded mode.

**The “degraded” mode for states.** The functionality “history service in degraded mode for states” shall provide the correct working of the newly to be implemented history service in the degraded mode with the different modes “full”, “partial” or “no” history for states. If “state rollback” is the preferred constraint reconciliation mechanism it shall store a history entry with a higher version than the last existing history entry version. If “full history” is chosen, there are no limits of history entries. The “partial mode” shall be configured via a memory parameter and defines the maximum number of history entries stored within the degraded mode.

**The “healthy” and “degraded” mode for state history.** The history service shall provide the following functionality within the two modes “healthy” and “degraded”: If entities are created during the “healthy” system mode, the replica table shall be filled with a new replica. Updates during the healthy mode shall lead to an updated replica value. The DeDiSys middleware bypasses the history service during the “healthy” mode. If the system mode switches to “degraded” mode and an update occurs to an object two new history entries shall be created: One with the tag “constraint consistent” representing the last version of the replica during the healthy mode, and one with a tag “degraded” for the newly updated version. If updates occur during the degraded mode other entries shall follow with the tag “degraded” to the history state table.

**History service in constraint reconciliation phase - state rollback.** The history service shall provide the ability to deliver correct data during constraint reconciliation. For the “state rollback” it shall be a number of history entries from the degraded mode and the last reconciliation state (constraint-consistent, if the last reconciliation phase was finished correctly and, replica-consistent if not). Additionally, the states have to be delivered in a well defined order (see Section 4.4.3).

**History service in constraint reconciliation phase - operation compensation.** For the “operation compensation” mode a threat is coupled via a unique identifier to a distinct invocation operation, e.g. the method addBooking() from the flight booking test application (see Section 5.1). Each application may provide compensating classes which are used during the constraint reconciliation phase, if an according operation has created a threat. The external flightbooking application shall provide the correct behavior from calling the corresponding classes within the application till the update and persisting of the state within the DeDiSys framework and the application.

### 4.1.3 Functional Design of the History Service

The functional design of the history service is illustrated here:

**The interface of the history service.** It is common practice in the software industry to decouple system components to facilitate the design and implementation phase. Hence, interfaces between such components are necessary: The interface of a software module is kept separately from the implementation of that module. While the interface contains the signature only, the implementation provides the implemented logic of the method signatures. Such a mechanism is provided in Java with the different constructs Class and Interface.

As declared in the motivation and requirements, the history service shall be usable for a number of components within the DeDiSys EJB framework - such as Replica or CCMgmt (the constraint consistency manager). Hence, an interface IHistory has been introduced which characterizes the provided functionality.

As shown in Figure 7, there are five different operational groups within the interface:

- State history: These are several methods regarding the administration of state history. deleteCompleteHistory() deletes the complete history of all replica objects and shall be used, if all conflicts are solved and history items remain in the database (this also affects entries in the operation history table). deleteStateHistory() provides the deletion of all history states for a given replica. This functionality is used, for example, during the constraint reconciliation when a new state is chosen. If a replica object is deleted during the "degraded" mode, the history remains and thus the delete call for the replica has no effect at the same time. storeStateHistory() guarantees the storage of a new version of a replica, regarding the existing version and therefore it increments to a higher version. Finally, loadStateHistory() is the choice if history state with a distinct version shall be loaded.

- Operation history: The method deleteOperationHistory() requires a unique invocation identifier (uid) (see 4.1.3) as parameter. This functionality is needed after an external successful compensation of the stored operation history has happened. As for state history, storeOperationHistory() stores a distinct method call to the operation history database. loadOperationHistory() loads a set of saved items regarding the stored operation.

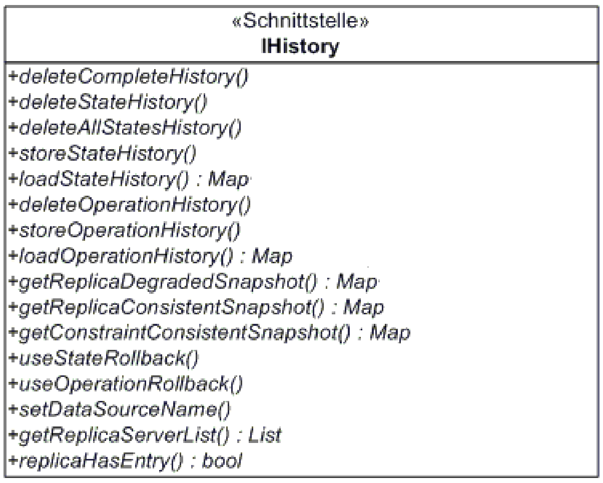

«Schnittstelle»
IHistory
+deleteCompleteHistory()
+deleteStateHistory()
+deleteAllStatesHistory()
+storeStateHistory()
+loadStateHistory() : Map
+deleteOperationHistory()
+storeOperationHistory()
+loadOperationHistory() : Map
+getReplicaDegradedSnapshot() : Map
+getReplicaConsistentSnapshot() : Map
+getConstraintConsistentSnapshot() : Map
+useStateRollback()
+useOperationRollback()
+setDataSourceName()
+getReplicaServerList() : List
+replicaHasEntry() : bool

Figure 7: History service interface

- Snapshot: An essential part of the history service is the possibility of loading a snapshot out of the database: getReplicaDegradedSnapshot() returns a map ordered by the version of the complete history for a distinct replica which is stored in the "degraded" mode. The methods getReplicaConsistentSnapshot() and getConstraint-ConsistentSnapshot() return a map ordered by the version of the complete history for a distinct replica. This replica is either "replica-consistent" or "constraint-consistent". Replica-consistent means that it is stored directly after replica reconciliation has ended. On the other hand, constraint-consistent replicas are stored after constraint reconciliation has ended or the systems enter the degraded mode and an update occurs for the first time in the "degraded" mode. Then a previous - constraint-consistent - version of the replica needs to be saved.

- Parameters: Three methods are available for setting parameters within the history service: setDataSourceName() for setting the data source name, useOperationRoll-back() for storing a boolean value (if operation compensation is allowed) and useS-tateRollback() for configuration (if the system shall use state rollback). Important is that either operation compensation or state rollback is allowed. If both values are set true, state rollback is the first choice, as it is the primary intention of the DeDiSys EJB middleware.

- Some other issues are provided via the history service interface. getReplicaS-erverlist() returns the database entry "partitionelements" with the highest version of a distinct replica. getReplicaListWithHistory() returns a set of object identifiers of all replicas which have at least one history entry. Finally, replicaHasEntry checks, if a given replica object has an entry in the state history table.

The provided methods are used from different modules from inside and outside (outside means external applications like the "History Application"; see Section 4.5 for further details) according to the needs of the individual software components.

**Configuration of the history service.** Due to the fact that the DeDiSys platform is an integrated middleware, different applications can make use of this kind of software. Hence, several needs regarding functionality of the history service might arise. So the possibility of configuring the history service to its individual needs has been implemented within this masters' thesis:

It is common practice within the DeDiSys project (e.g. see (Horehled, 2006) for the configuration of the constraint consistency management) to configure software using extended meta language (XML) files. The structure of these files is arbitrary, as the parsing functionality has to be integrated by the provider of the XML file. The configuration file of the history service is named history.xml. Listing 1 provides a sample configuration.

Listing 1: history.xml

```xml
<?xml version=''1.0''?>
<history>
  <!-- define if states or operations rollback is used with yes | no
      -->
  <!-- if state based rollback has value ''yes'', no operation based
      -->
  <!-- rollback is possible -->
  <rollback_mode>
    <state_rollback>yes</state_rollback>
    <operation_rollback>no</operation_rollback>
  </rollback_mode>

  <!-- allowed modes are full | partial | no-->
  <!-- full mode saves all states for a given replica-->
  <saves>full</saves>

  <!--configure partial mode-->
  <partial_mode_config>
    <!-- memory defines the number of states saved for a given
        replica; overwrites older states in FIFO-way-->
    <memory>5</memory>
  </partial_mode_config>
</history>
```

This file is read at the startup of the JBoss server. Modifications are only possible after shutting down the system, changing the parameters and restarting the server. At first, the user may choose between state rollback (Section 4.4) or operation compensation (Section 4.3). Only one of them is possible. Next, the modes of the history service have to be defined: "Full" stores all operation and states (incrementing the version by one after each save for a given replica) as the history service is used. For "partial" an additional memory parameter has to be set. If the number of entities for a given replica exceeds the memory parameter, the oldest entry with the tag "degraded" is deleted. "No" history does not save history entries and is not intended for a DeDiSys system which wants to compensate or rollback. If "full" history is chosen the store function of the service is not limited with respect to older stored objects. Theoretically, the snapshot function is slowed down due to a high number of history entries because a huge number of updates are made during the degraded mode. However, if "partial" is the choice the number of entries of a distinct replica have to be found out at each database access - with a guaranteed delete for each entry for a massive number of updates. Summarized, in theory "full" provides faster storing, but slower reading, whereas "partial" provides slower storing and faster reading in the long run. The parsing function of the history.xml is implemented in the History file, the core business logic of the service.

**The history service SessionBean.** On the one hand, there are the history service core function and its defined interface. On the other hand, there has to exist a "glue-logic" where external client software like the "History Application" might use the history service itself. Thus, a "Session Bean" has been used for this purpose. For a general introduction to session beans see Section 2.1.1. The design of the history session bean is comparable to the ccpersistence session bean described in (Horehled, 2006), as both are core functionalities of the DeDiSys middleware. In Figure 8 one can see the structure of the implemented session bean: Three different components build up a session bean:

- Session bean's home interface: The interface HistorySessionHome extends EJBHome and provides a create() method which matches the create() method in the enterprise bean class.
- Session bean's remote interface: HistorySession extends EJBObject. A remote interface is the client's view of the bean and the defined functions within are visible to the client application. So, the methods illustrated in Figure 7 are found within this type. For all these methods there exists an implementation in the enterprise bean class. Additionally, a few methods like for example getReplicaByObjectid(), which make use of the ReplicationManager, shall support the needs of the used client components.
- Session bean's enterprise bean class: In HistorySessionBean, which implements SessionBean, the business logic of the bean is provided. Each method call of the session bean's interface (session bean remote) forwards the call to the core history service. Thus, it provides a decoupling of the history service and external applications. However, not only the remote interface methods must be implemented. Methods from the SessionBean interface also require some lines of code: ejbRemove() (instance is being removed by the container), ejbActivate() (instance is activated), ejbPassivate() (instance is passivated) and setSessionContext() (to pass a reference to the SessionContext to the bean instance). For further details see (Roman et al., 2005).

The communication of the client with the session bean (and furthermore with the history service) is shown in Figure 9. Home and remote interfaces of the session bean provide a clear decoupling.

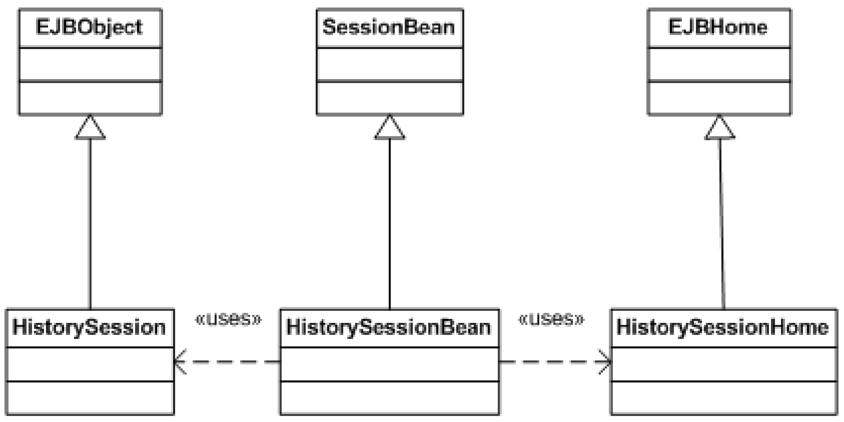


Figure 8: History Session Bean with its different components

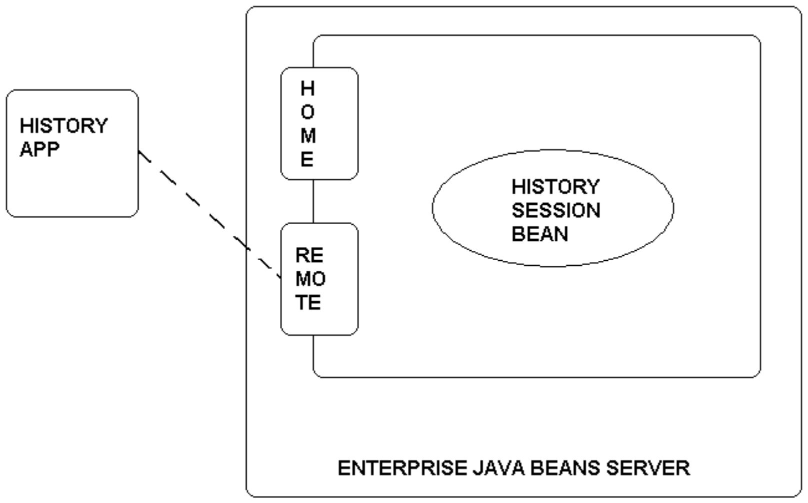


Figure 9: History Session Bean communication

To create the file historyservice.jar, an EJB deployment descriptor for JBoss is necessary: It is the job of the JBoss application server to administrate the different session and entity beans. Therefore, each bean requires a file named ejb-jar.xml. The history specific one is partially provided in Listing 2: The bean type must be declared at the beginning of the file (session). Then the above described interfaces and implementing classes have to be written, just like whether if it is a stateless or stateful session bean (here: stateless).

Listing 2: ejb-jar.xml for the history session EJB

```
<?xml version=''1.0'' encoding=''UTF-8''?>
<ejb-jar version=''2.1'' ... >
  <display-name>develop</display-name>
  <enterprise-beans>
    <session>
```

```
      <display-name>HistorySessionEJB</display-name>
      <ejb-name>HistorySessionEJB</ejb-name>
      <home>org.dedisys.history.session.HistorySessionHome</home>
      <remote>org.dedisys.history.session.HistorySession</remote>
      <ejb-class>org.dedisys.history.session.HistorySessionBean</ejb
         -class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>

  ...

</ejb-jar>
```

As illustrated in (Horehled, 2006) the following way is possible for the modification of JBoss specific deployment descriptors: A context-specific descriptor named jboss.xml is provided with the bean. So the history service makes use of its own jboss.xml file, presented in Listing 3.

Listing 3: jboss.xml for historyservice.jar

```
<?xml version=''1.0'' encoding=''UTF-8''?>
<!DOCTYPE ... ''>
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>HistorySessionEJB</ejb-name>
      <jndi-name>HistorySession</jndi-name>
      <local-jndi-name>HistorySessionLocal</local-jndi-name>
    </session>
  </enterprise-beans>

  ...

</jboss>
```

**History service MBean.** History service via an MBean is an alternative approach to the usage of the session bean. An introduction into MBeans can be found in (Horehled, 2006) and Section 2.3.2. In short, MBeans are Java objects that encapsulate a resource and expose it for management. Via MBeans one can provide an interface to steer its application. For the history service a standard MBean has been used. Figure 10 presents the different components necessary to create the service: The management interface is defined in HistorySvcMBean which extends the ServiceMBean. As second step the HistorySvc represents the service implementation which implements HistorySvcMBean and ObjectFactory (according to (Shannon, 2003), the JNDI framework allows for object implementations to be loaded in dynamically via object factories). Thereto, it extends the JNDIAllocatableSvc, a DeDiSys internal MBean support class. Within HistorySvc, the singleton of History gets called and therefore allows the usage of all methods there.

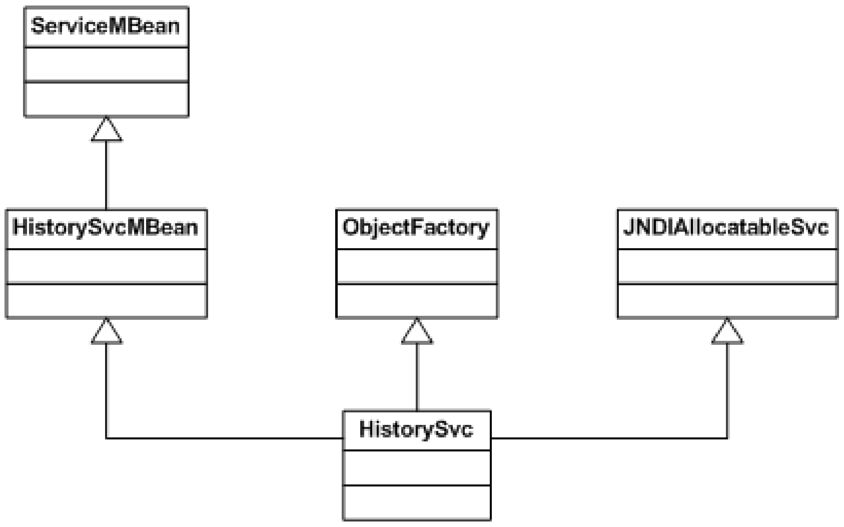


Figure 10: History MBean with the different modules

The deployment descriptor of this MBean can be found and is presented in Listing 4. It contains the attribute name, as the naming service of JBoss needs a unique identifier to list it.

Listing 4: Part of the jboss-service.xml

```
<mbean code=''org.dedisys.history.mbean.HistorySvc'' name=''dedisys.
    mbeans:service=History''>
  <attribute name=''JndiName''>inmemory/HistoryService</attribute>
  <depends>jboss:service=Naming</depends>
</mbean>
```

The usage within the DeDiSys project is reached with HistoryFactory, a class accordant to the factory-pattern.

**History service interceptor.** As defined in the requirements, the history service shall store operations which are triggered through external applications. Thus, a mechanism is needed to “catch” such operations on EJB objects. Operations are method calls with arbitrary arguments. The JBoss application server provides a so called interception module which allows to add additional interceptors as “filters”. Such filters are combined in an interceptor chain. The distinct position in the interceptor chain is important, as it defines the preceding and following interceptors. Due to the fact that interceptors are able to modify calls, the placement of new interceptors is not arbitrary. Each entity bean type requires its definition for the interceptor chain. They are provided in the file standardjboss.xml. Listing 5 shows an entry regarding the standard entity bean with container-managed persistence for the history service interceptor with its preceding and following interceptor. For further details on the interception framework of JBoss see (Horehled, 2006).

Listing 5: Part of standardjboss.xml

```
<container>
   ...
   <interceptor>org.jboss.ejb.plugins.
       EntitySynchronizationInterceptor</interceptor>
   <interceptor>org.dedisys.history.HistoryInterceptor</interceptor>
   <interceptor>org.eu.adapt.bs.compmonitor.ejb.AdaptInterceptor</
       interceptor>
</container-interceptors>
```

Part of the work related to this research was to implement a HistoryInterceptor. This class extends the AbstractInterceptor of the JBoss server. There are two methods of interest within the HistoryInterceptor: The first method is the invoke() method. It is the entry point in the interceptor and solves the issue of filtering out operations of interest. If the system is in "healthy" mode no operations will be saved - therefore the call is forwarded without any modifications. If the system is in "degraded" mode and the calling method fulfills the following requirements, operation is stored in the operation database table:

- The invoking operation must be a top-level invocation within the EJB container. Within this project top-level invocations appear only from the session bean of the flight booking application (FlightBookingFacadeBean), as the business operations which execute corresponding changes to the Entity Beans are implemented there.
- Configuration in the history.xml must allow operation compensation.
- The called method must have a "compensating method" as it defined in the CompensationClasses, see Section 4.3.3 for further details.

The second method of interest is the parseMethod(). It handles the parsing of the operation name. When the requirements are fulfilled, the invocation object — which is handed from interceptor to interceptor — gets set two values of interest: A unique invocation identifier and a list in string format this way representing all reachable nodes of the system during this very invocation call. Finally, the invocation is stored in the operation history table with storeOperationHistory().

**History service core design.** The history service core functionality is included in the History class. It implements the IHistory interface, thus providing the functionality of the requirements. The singleton pattern is used to guarantee the usage of only one instance within the memory. This is useful, as many object calls from different classes require database access to the underlying tables "history" and "operationhistory". These tables contain all necessary information in order to provide state rollback and operation compensation:

The “history” table is used for states and contains of the following attributes:

- objectId: This is the unique identifier of the history state entry and accords to the object identifier within the ComponentHandle of the replica.
- entry: This is the state of the replica object and is stored with the sql-type BLOB.
- entry_type: Entry_type was used in prior approaches where operations and states were stored in one database and had to be distinguished by this attribute. At this point, this is deprecated.
- version: Whenever a distinct replica is updated, its version is incremented by one — to differentiate the states and allocate them in temporary order.
- tag: The “tag” attribute has three different entries: “Degraded”, “constraint-consistent” and “replica-consistent”, see Section 4.1.3 for further details.
- partition-elements: When a history entry is stored in the database, the MembershipManager is called for a list of reachable nodes. This list is stored, as it helps to speed up the reconciliation process (described in Section 4.2.3).

The “operation history” table is used for operations and consists of the following attributes:

- uniqueId: This attribute is the unique identifier stored in the invocation object.
- method: This is the method name stored in the invocation object.
- arguments: The arguments of the methods are also stored in the invocation object and necessary for compensating this action and stored as sql-type BLOB.
- class: Attribute “class” represents the calling class for the invocation.
- partitionelements: When a history entry is stored in the database, the MembershipManager is called for a list of reachable nodes. This list is stored, as it helps to speed up the reconciliation process (described in Section 4.2.3).
- objectId: If the invocation object has an identifier it is stored here.

Both table structures are provided with the file structure-mysql.sql within the DeDiSys sub-directory “sql”.

An alternative approach would be that history entries in the table could be entity beans instead of the ones in the flight booking application (FlightBean, BookingBean, PassengersBean), which are stored in dynamically created database tables. However, now all entity beans in the DeDiSys-enabled JBoss are replicated. For example, the configuration about replicas is stored locally and does not get replicated. A configuration of replicated and not replicated entity beans and which replica reconciliation handler is used for distinct entity beans is not possible at the moment in the replication protocol.

Most of the methods shown in Figure 7 make use of the underlying database. Access to the tables is reached with the objects PrepareStatement and Connection. The usage of other tools like Hibernate, which is an open-source persistence framework for Java, has not been taken into consideration. This framework allows to persist and read out the state of an object to/from a relational database like MySQL.

## 4.2 Replication Protocol in DeDiSys

In the section about fundamentals, the theoretical aspects regarding replication protocols Section 3.2 are introduced. Within the following lines, enhancements to the existing DeDiSys EJB replication protocol shall be illustrated.

### 4.2.1 Motivation for a new Replication Protocol

In the past, a primary-per-partition protocol (P4) with a relaxed passive replication model has been implemented for EJB within the DeDiSys research project. It allows partitions to continue, when nodes are separated due to network failure. Now the P4 relaxes consistency during partitioning, but full consistency shall be restored at reconciliation time. A detailed description can be found in (Beyer et al., 2005).

One of the first activities of the protocol in the reconciliation phase is missed update propagation. With such a mechanism the different replicas are propagated between the different nodes to get a common base of existing replicas. Unfortunately, the existing protocol uses one message for one replica per multicast. This is no problem when the number of replicas is rather small (such as in a test environment), but during normal operations this slows down the duration of the reconciliation. Additionally, the giving solution propagates replicas even if no updates happened during the degraded mode.

### 4.2.2 Requirements for a new Replication Protocol

The requirements regarding the new replication protocol in the DeDiSys EJB environment are the following:

**Improved DeDiSys replication protocol - basic functionality.** The improvement of the DeDiSys replication protocol shall be implemented in a new class, derived from the pre-used DeDiSys protocol. It shall hold the required basic functionality as defined in (Froihofer (ed.), 2005a).

**Combined updates.** The existing replication protocol allots propagation of missed updates at the beginning of reconciliation time. Each replica is sent separately to the other reachable nodes over the network. To enhance the performance of the missed update propagation, several replicas shall be combined to have less update propagation messages.

**Avoidance of updates.** The given solution enforces updates on replicas of partitions, even if there have not been any updates on them during their last degraded phase. The enhanced protocol is required to verify if missed update propagations are necessary and provide them in case of an update.

**Replication Manager.** The ReplicationManager, which interacts with the replication protocol, needs enhancements regarding the distribution of history states. Normally, it is not intended to distribute all history states, as they shall be stored locally due to the amount of memory they require. But history states with the tags "replica-consistent" or "constraint-consistent" need to be distributed to all reachable nodes.

### 4.2.3 Functional Design of the new Replication Protocol

The functional design of the new replication protocol mainly required changes in two classes: The ReplicationManager and the newly created DedisysHistoryBasedProtocol. As shown in Figure 4 on page 16, the replication manager and the replication protocol shall be decoupled components.

**DedisysHistoryBasedProtocol.** The DedisysHistoryBasedProtocol has the file DedisysProtocol as a basis. An entry point for performance enhancements is startReconciliation(), which is called method from run() (from the inner class ReconciliationStarter of the ReplicationManager), marking the beginning of the reconciliation phase. The propagation of missed updates takes place there. Instead of sending each replica with its own transaction, several replicas are combined in a list. The size of the list depends on the object size of the replica (measured via serialization) and is a configurable parameter for test reasons and configuration for application needs within the protocol. The method setMessageSize() allows the external setting via the control application of the parameter message size. Thus, the number of needed messages is reduced, which lowers the network usage and therefore speeds up the missed update propagation.

Another enhancement is the method checkIfNotChanged(): At first, the history service provides a list of reachable nodes of the highest history version for a distinct replica. If the list equals the list of actually reachable nodes no missed update propagation will occur for this replica, as all involved partitions share the same highest history state and therefore the same replica.

In Figure 11 the interaction between the replication manager and the improved protocol is depicted.

**ReplicationManager.** Several modifications were necessary to adapt the class ReplicationManager according to the tasks given for this thesis: At first, the history service is usable within the manager, as the configuration read-in of the history service occurs at the startup of the ReplicationManager within the method init(). The methods finishReconciliation() and call() are now able to store "constraint-consistent" and "replica-consistent"

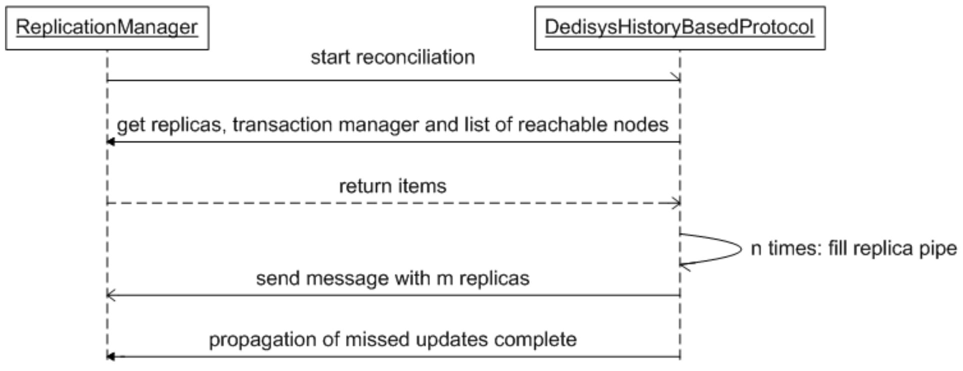

ReplicationManager
DedisysHistoryBasedProtocol
start reconciliation
get replicas, transaction manager and list of reachable nodes
return items
n times: fill replica pipe
send message with m replicas
propagation of missed updates complete

Figure 11: Sequence chart for replication manager and improved protocol

replicas, as there exists a differentiation mechanism (via tagging) to associate replica states to system modes.

The ReplicationManager on one node communicates with messages with the managers on other nodes to fulfill different tasks. Namely the creation and deletion of components, starting and stopping of reconciliation and so on. It provides a message handler and is able to differentiate between various message types. The messages itself are classified by MessageType. So a more comprehensive enhancement was the integration of an enhanced messaging behavior. This was done in the processMessage() method: The method processes objects of the type PropagationMessage which are sent between the nodes for communication reasons. These reasons are stored as parameters in the PropagationMessage object. Thus, the base class needs to be adapted for defining more message types. Within the ReplicationManager, for each message type an according method for handling is implemented. The enhancements implemented within this thesis' project are:

- GET_DEGRADED_STATE_HISTORY: This returns the "degraded" history of a given replica identifier from the local or remote node.
- GET_REPLICA_CONSISTENT_STATE_HISTORY: A "replica-consistent" history of a given replica identifier from the local or remote node is returned.
- STORE_REPLICA_CONSISTENT_HISTORY: This stores the replica with the help of the history service with the value "replica-consistent" for the tag attribute in the database.
- STORE_CONSTRAINT_CONSISTENT_HISTORY: Similar to the previous one, this stores the replica with the help of the history service with the value "constraint-consistent" for the tag attribute in the database.
- DELETE_STATE_HISTORY: This deletes the history object with the "constraint-consistent" and/or "replica-consistent" tag. History entries with the "degraded" tag are not affected, as this message is needed for internal updating of history entries.

The configuration of the replication manager and replication protocol is done with the file compmonitor-config.xml, a sample of which is shown in Listing 6. During this project this file and therefore the parsing functionality within the ReplicationManager has been enhanced according to the requirements. Now it is possible to choose an arbitrary replication protocol (which must implement the interface IReplicationProtocol within the tag <dedisys-protocol>). So, several protocols can be tested and evaluated with an easy configuration. Another feature has been added: In the prior solution only one reconciliation handler was allowed to be used during a ready-to-use system. Now, a number of them can be defined within the file, under the usage of the <reconciliation-handler> tag. The last entry accords to the default one in the ReplicationManager. With the DeDiSys control application other ones can be chosen as long as they are defined in the compmonitor-config.xml, see the technical report (Ertl and Ji, 2006) about the utilization of replica reconciliation handler in the EJB middleware.

Listing 6: compmonitor_config.xml

```
<?xml version=''1.0''?>
  ...
<componentmonitor code=''org.dedisys.replication.impl.
    ReplicationManager''>
  ...
  <datasource>java:/ReplicationDS</datasource>
        <dedisys-protocol>org.dedisys.replication.impl.
            DeDiSysHistoryBasedProtocol</dedisys-protocol>
        <reconciliation-handler>org.dedisys.replication.impl.
            PrimaryWinsReconciliationHandler</reconciliation-handler>
        <reconciliation-handler>ejb.ReplicaReconciliationFlight</
            reconciliation-handler>
        <reconciliation>
                <reconciliation-handler>org.dedisys.replication.impl
                    .ConfigurableReconciliationHandler</
                    reconciliation-handler>
                <param-name>reconciliation-host</param-name>
                <param-value>128.131.81.13</param-value>
        </reconciliation>
        <server weight=''1''>128.131.81.13</server>
        <server weight=''1''>128.131.81.14</server>
  </segment>
</componentmonitor>
```

For further details concerning the ReplicationManager, see (Künig (ed.), 2007).

## 4.3 Operation Compensation

Operation compensation as part of the reconciliation process is illustrated in this section.

### 4.3.1 Motivation for Operation Compensation

To restore consistency within the DeDiSys middleware, the process is split into two different phases: The first phase is the replica reconciliation process, described in Section 4.2, and the second one is the "constraint reconciliation". DeDiSys follows two different strategies in the EJB framework to reach a valid state: On the one hand, there is operation compensation, on the other hand state rollback. This section discusses the operation compensation.

Transaction managers, such as the JBoss internal one, may roll back active transactions but not committed ones. Operations which are stored during the degraded mode are already committed. During the compensation phase, the database is accessed to read out distinct information of such committed and stored operations. Thus, the following rollback to a previous state is a new transaction and must not be confused with the committed ones from within the degraded mode.

The compensation is realized via an external application, ideally the application which invoked the operation on a distinct entity bean. So the mechanism to create a constraint consistent state is not part of the DeDiSys framework, but the reconciliation process is controlled by the middleware. It is assumed that the application developer knows best how to solve a distinct conflict according to the intention of the application itself. In contrast, the DeDiSys framework has only the possibility to provide a generic approach (state rollback) which may not fit well for all software modules which make use of the middleware.

It is worth mentioning that the previously described history service is the basis for such a behavior. The persisted information of the operation is presented to the external application: So, a tailored reaction to guarantee constraint consistency is possible.

Another research approach within the DeDiSys project aims at a redo of all operations within a sandbox environment, see Section 3.2.2 for further details.

### 4.3.2 Requirements for Operation Compensation

The operation compensation process is split into a DeDiSys part and a part for external applications. This subsection handles the requirements for both of them:

**Operation compensation and the history service.** The user has to choose if operation compensation or state rollback shall be used as a constraint reconciliation mechanism. Thus, the history service allows a configuration to support the approach of operation compensation. The history service persists distinct top level invocations which are method calls on entities with additional information during the degraded mode. Top level invocations are invocations from a ComponentMonitor which is not called from other ComponentMonitor instances. Thus, an invocation is not at top level if the origin monitor is reentrantly driven by a call from another ComponentMonitor. The compensating process shall access this data items to use them as decision support for constraint reconciliation.

**Process of operation compensation.** The process of constraint reconciliation via compensation of methods shall be controlled by the DeDiSys middleware. Existing modules of the framework shall therefore be enhanced and/or newly created.

**Compensating the operation.** If the DeDiSys system is configured for operation compensation, an external application shall provide the compensating method. To guarantee a generic approach for calling the compensation mechanism, the DeDiSys middleware shall provide a "compensation" interface which has to be implemented by the external application. In any case, the result of the compensation shall be returned to the framework.

**Registration of compensable methods.** The developer of the external application is responsible for the registration of methods which shall be compensable. Within the DeDiSys middleware such a registration mechanism for external applications shall be provided.

### 4.3.3 Functional Design of Operation Compensation

The functional design of the operation compensation will be presented in two parts: At first, the software modules in the DeDiSys EJB framework are illustrated. Afterward, the compensation modules in a specific external application — the flight booking application — are described.

#### Operation Compensation in DeDiSys

As mentioned above, the DeDiSys middleware provides the control module of the operation compensation.

**Compensation mechanism within the constraint consistency management.** The reconciliation mechanism is integrated within the constraint consistency package. Briefly described, the compensation mechanism works like the following: If the history.xml is configured for operation compensation as illustrated in Listing 1, all top level invocations are saved in the "degraded" mode to the operation table which implements the ICompensation interface. After the replica reconciliation phase, the DeDiSys middleware initiates the compensation mechanism. Therefore, a ReconciliationManager and ReconcilisationSessionBean have been introduced in prior work, see (Fuchshofer, 2006).

The ReconciliationManager now uses methods of the history service to differentiate between operation compensation or state rollback as further strategy. The method of interest is reconciliationForApplication(), where reconciliation for the given application is performed. It retrieves all accepted threats from the module CCPersistence which is responsible for storing and retrieving consistency threats. Then the method from the ReconciliationSessionBean solveThreatByCompensation() is called. Finally, the threat is deleted from the threat database. The call to the ReconciliationSessionBean is packed within a transaction of JBoss, a rollback is possible which is an atomic process.

The method solveThreatByCompensation() provides the following functionality: It reads the invocation unique identifier out of the stored consistency threat and assigns and requests the history service to retrieve the stored operation. One of the stored parameters is the operation name. This is used to regain the compensation class as an object out of the CompensationClasses data structure. Finally, the compensating method of the external application is called and the value of the boolean return parameter indicates if the threat was solved by the application. Then the threat and operation history will be deleted. Otherwise, a hint is given to the operator that the compensation for the given threat has failed.

In Figure 12 the interaction of different modules for operation compensation is depicted.

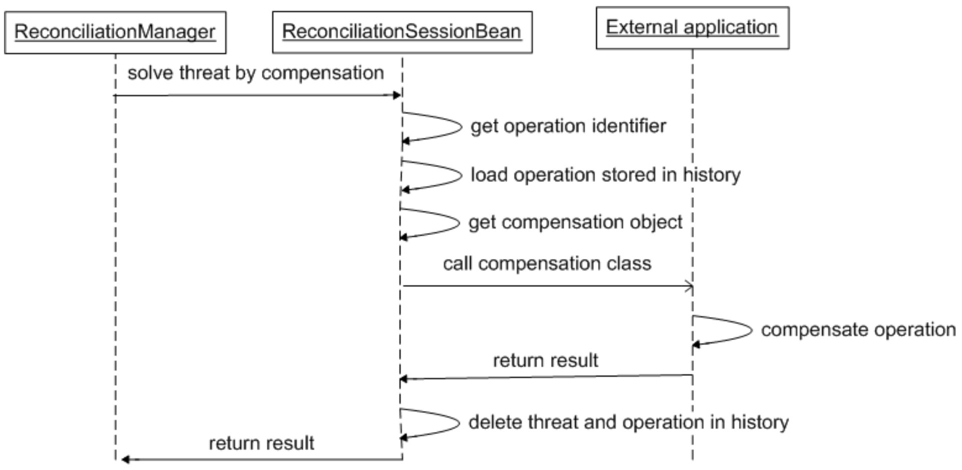


Figure 12: Sequence chart for operation compensation

**The ICompensation interface.** The adapted middleware now provides an ICompensation interface, which must be implemented by a class in an external application to allow operation compensation. Each bean that causes a top-level invocation (like session beans) may implement this interface. So, the history service can store an entry in the operation table, whenever a method in the bean is intercepted, see Section 4.1.3 for further details. In Listing 7 this new interface is shown: The first parameter operationMethod represents a string of the method which shall be compensated. Additionally, the parameter arguments is needed. It provides the distinct argument values of the method in an object array. Thus, constraint-consistency for the given entity is possible, as the exact data items are restored from the history database. The implemented method returns the boolean value true, if the compensation was successful. Thus, the entity is constraint consistent. If it returns false the application cannot immediately compensate the operation. A reason for this may be that the application has not implemented all cases regarding the argument values.

Listing 7: ICompensateOperation.java

```java
public interface ICompensateOperations {

    public boolean compensate(String operationMethod, Object[]
        arguments);

}
```

Within this project, this interface is implemented in the flight booking application as a proof-of-concept where the documented behavior is executed.

**Compensation import.** As the middleware allows any external application to use the operation compensation mechanism for their own compensation classes, a definition of classes that shall be compensable must be provided for the DeDiSys framework. Hence, a generic document structure according to the document type definition (DTD) of XML is introduced. A document type definition (or schema-definition) is a set of rules to represent documents of a given type, and a document type is a class of similar documents (like phone books). Summarized, in a DTD the order, the nesting of elements and the kind of content of attributes are declared. The implementation of such a document structure takes place in an XML file.

The file comp_def_1.0.dtd demonstrates such a DTD structure in Listing 8:

Listing 8: comp_def_1.0.dtd

```xml
<?xml version=''1.0'' encoding=''UTF-8''?>
...
<!ELEMENT compensate (threat-compensator+)>
<!ELEMENT threat-compensator (context-class, method)>
<!ELEMENT context-class (#PCDATA)>
<!ELEMENT method (#PCDATA)>
```

The <!Element ...> item represents a tag attribute, in which a number of parameters are allowed. An example of a compensation file is compensate.xml of the flight booking application, illustrated in Listing 10, with a detailed description of the tags.

As the implementation of the DTD structure is an application developer's responsibility, the DeDiSys middleware itself needs a parsing mechanism to gain knowledge of all methods which shall be compensated. This concerns external applications at JBoss-startup which are unknown to the DeDiSys middleware. Thus, the CompensationImporter is used. Additional modifications are necessary in the files XMLHelper and ValidatingXMLParser in the constraint consistency management package of DeDiSys to embed the correct parsing behavior.

### Operation Compensation in an Application

The matching part of the DeDiSys middleware during constraint reconciliation with operation compensation is the compensating class in the application. During the implementation process of this thesis, the flight booking application was used as proof-of-concept for operation compensation, see Section 4.6 for an introduction to the flight booking application.

**The implementation of the compensation class.** Compensation of methods in an external application is tied together with constraints regarding the application: There are a number of constraints defined for the flight booking application. The middleware requires a ccDefinitions.xml file for each application which makes use of DeDiSys. In this file, all constraints of the application are defined with a specific structure, see (Horehled, 2006) for further details on constraint definition. One of the constraints in the flight bookings' ccDefinitions.xml is of special interest for this thesis: It is called "PartitionSensitive", a "HARD" constraint with "REGULAR" priority, and shown in Listing 9. This constraint defines a total number of passengers within one flight for the economy class. This number is identical to the seats available in the plane. During the "healthy" mode it is not allowed to overbook the flight, while in "degraded" mode an overbooking is possible. The reason is a tradeable constraint, which is defined for the application. Such tradeable constraints are relaxed during the "degraded" mode to allow availability of the system. As different partitions exist, it is not guaranteed that this constraint is fulfilled, thus it is temporarily relaxed, see Section 3.1.2. According to the overbooking a threat is stored in the table ccpersistence with the help of the constraint consistency management and constraint reconciliation is necessary. The complete constraint consistency management is described in (Horehled, 2006) and (Fuchshofer, 2006).

Listing 9: Constraint "PartitionSensitive"

```
<constraint name=''PartitionSensitive'' type=''HARD'' priority=''
    REGULAR'' negotiation=''IMMEDIATE'' contextObject=''Y''
    minSatisfactionDegree=''POSSIBLY_SATISFIED''
    latestAcceptedSatisfiedThreatRemovesIdenticalThreats=''Y'' intra-
    object=''false''>
  <class>Constraints.flightpartitionsensible</class>
  <context-class>ejb.FlightBean</context-class>
  <expression>flight.flightNotOverBooked</expression>
  <affected-methods>
    <affected-method>
      <context-preparation>
          <preparation-class>org.dedisys.ccmgmt.
              CalledObjectIsContextObject</preparation-class>
      </context-preparation>
      <objectMethod name=''setBookedSeatsEC''>
        <objectClass primKeyField=''FlightID''>ejb.FlightBean</
            objectClass>
        <arguments>
          <argument>java.lang.Integer</argument>
        </arguments>
      </objectMethod>
    </affected-method>
  </affected-methods>
</constraint>
```

One of the core functionalities of the flight booking application is the addBooking() method in the FlightbookingFacadeBean. So, an arbitrary number of passengers for business class and economy class is added to a flight, as long as the total number of booked seats is not reached. Now it is assumed that the current booking process for the economy class would exceed the total number of seats. Operation compensation is the chosen compensation mechanism in the middleware. During the "healthy" mode the middleware will reject the call, as the constraint consistency management identifies the overbooking. In the "degraded" mode, the HistoryInterceptor identifies the method as a compensable method (as the compensate.xml in the flight booking application registers it to the DeDiSys framework) and triggers the history service to store the operation in the operation table. Additionally, the method call of addBooking() is executed as if there were no constraints. When the nodes are connected again, the constraint reconciliation process illustrated in Section 4.3.3 calls the responsible method:

CompensateAddBooking is the compensation mechanism in the flight booking application. As it implements ICompensateOperation, it has a compensate() method as the core business logic. The parameter array of the restored operation is used for the following procedure: At first, the booking process is tried out again. If the flight booking internal reconciliation handler replicareconciliationflight returns a useful state, the compensation is successful. If the previous booking approach does not work, all the other flights with the same from/to/departure parameters are attempted as booking flights. Finally, if neither step turns out to be successful a new flight is created and the original intended booking added.

As the method interacts with the constraint consistency management of DeDiSys, a CompensationNegotiationHandler and NegotiationHandlerContainer are necessary to negotiate specific actions with the CCManager. Basically, the negotiation handler returns always "true" in the method isReconciliationActivity() — in contrast to the DummyNegotiationHandler which always returns "false". Hence, all existing threats within the operation compensation are accepted. This is correct, as the applications may have an implemented behavior which overrules constraints. Further infos concerning the negotiation and negotiation handling can be found in (Fuchshofer, 2006).

**Usage of compensate.xml.** As described in Listing 8 each application needs to implement the file comp_def_1.0.dtd for all methods that shall be compensable. Therefore, the flight booking application presents the compensate.xml file. In Listing 10 one can see the sample used in this thesis:

Listing 10: compensate.xml

```
<!DOCTYPE compensate SYSTEM ''comp_def_1.0.dtd''>
<compensate>
  <threat-compensator>
    <context-class>Compensation.CompensateAddBooking</context-class>
    <method>addBooking(dataVO.InvoiceVO,dataVO.FlightVO)</method>
  </threat-compensator>
</compensate>
```

The tag <threat-compensator> wraps the necessary information for the DeDiSys middleware:

- <context-class>: Here the distinct package as well as the class name of the class which compensates is needed.
- <method>: The method that together with its parameters shall be compensated is listed here.

## 4.4 State Rollback

An alternative approach to the previous described "operation compensation" is "state rollback".

### 4.4.1 Motivation for State Rollback

As illustrated in Section 4.3.1 about the motivation for operation compensation, the reconciliation process requires a strategy for the searching and setting of a constraint-consistent system state. While "operation compensation" requires external applications to fulfill this target, "state rollback" is a transparent process within the middleware. Thus, an external application using the middleware is not included in this process. As possible threat inconsistencies are negotiated during the "degraded" mode between the application and the middleware, the application is aware regarding constraint consistency because of the negotiation handling. However, no prototype implementation includes the application for state rollback activities. Thus, it is not necessary that applications store data about possible threats which are handled by the middleware.

### 4.4.2 Requirements for State Rollback

The requirements for state rollback are similar to those for operation compensation:

**Generic solution.** State rollback shall provide a generic rollback to prior states until a specific constraint is satisfied. The solution must not be set only to the last consistent state of a replica, as this would cancel all operations within the "degraded" mode. However, it is possible that due to a state rollback the last consistent state is the only remaining outcome.

**History service for state rollback.** The history service shall support state rollback with history states for replicas which are stored during the "degraded" mode. Additionally, the process of state rollback shall access the interface of the history service for the solution finding.

**Configurable behavior.** The constraint reconciliation shall be configurable to allow state rollback or operation compensation. Additionally, if state rollback is the choice different approaches for finding a solution shall be provided.

**Cleanup with the history application.** The solution provided by state rollback shall provide fast results. Afterward, it shall be possible that an external history application may change the states of affected replicas after the "healthy" mode (constraint-consistent mode) has been reached.

### 4.4.3 Functional Design of State Rollback

This subsection provides the functional design of state rollback.

**Reconciliation within the constraint consistency management.** As described in Section 4.3, the module for the constraint consistency management is the one for the constraint reconciliation process for historical reasons. There are two classes which are relevant for state rollback, the ReconciliationManager and the ReconciliationSessionBean.

Within the ReconciliationManager, the method reconciliationForReplication() is the entry point for state rollback. It performs reconciliation for the given application, retrieves all accepted threats from CCPersistence and calls solveThreat() from the ReconciliationSessionBean. This session bean performs the state rollback mechanism of the DeDiSys EJB middleware to reach constraint consistency. It has been implemented in prior work and is embedded within the constraint consistency management. The solveThreat() method is the main logic for constraint reconciliation. At first, the threat is attempted to be solved by the middleware. If this is not possible the application has to be informed that constraint reconciliation is not possible with the middleware. If the middleware finds a status of the affected objects that satisfies the threat, this solution will be presented to the application, which is asked for confirmation of this solution. Finally, a re-evaluation of the object is performed to check if the threat is really solved, as the application could have modified the objects again.

Several modifications are needed in the ReconciliationSessionBean to reach the aims of this work: To allow the distinction of different replica states, three private fields are integrated, namely DEGRADED_STATE_ VECTOR, the LAST_ REPLICA_CONSISTENT_ STATE_VECTOR and LAST_CONSTRAINT_CONSISTENT_ STATE_VECTOR. They map the data structures regarding the different modes where replica states are written in the history table. The build-up of these Vector classes is realized with external algorithms described in this Section 4.4.3.

- solveThreatByMiddleware(): Within the solveThreat() method, this piece of code tries to evaluate the given information stored in the ConstraintStatus and attempts to solve the conflict based on this information. If possible, this method takes the state of history objects (DEGRADED_STATE_ VECTOR, the LAST_REPLICA_ CONSISTENT_ STATE_VECTOR and LAST_CONSTRAINT_ CONSISTENT_ STATE_ VECTOR) in this consecutive way to find a solution. So, first the degraded states are tried out, afterward the replica consistent ones and finally the constraint-consistent state of the pre-degraded phase. At the same time it is intended that a state of the degraded mode can be used, as this would allow a number of operations that are not cancelled due to constraint reconciliation. The last consistent state clears all operations on each node of the degraded phase. Whenever the middleware finds a satisfying solution, the application is asked for confirmation of this solution (if requested by application). If the application confirms the solution the answer is "true"; "false" in any other case.

- checkCombinationsOfObjectStates: Here the interaction with the history service according to the actual state for rollback takes place and the states of the history tables of all nodes are requested. If it is possible that the degraded data structures are all empty it would mean that no operations are performed on the entities during the degraded mode.

- isThreatSatisfiedAndConfirmed(): This method is used within checkCombinationsOfObjectStates and sets the objects (identified by its handles like IComponentHandle to the states and verifies, if the constraint is satisfied.

**Detailed description of constraint reconciliation in the ReconciliationSessionBean.** The actual constraint reconciliation process of the reconciliation session bean with state rollback of the DeDiSys EJB implementation has not been documented since the work of (Fuchshofer, 2006). But the implementation of this reconciliation process has changed since then. Thus, the complete constraint reconciliation process is described here in detail.

In Figure 13 the state rollback is depicted.

The method solveThreat() in the ReconciliationSessionBean is the entry point for the constraint reconciliation. It is called for each threat which is stored in the database of the constraint consistency manager. A number of so-called "affected objects" belong to such a threat. In solveThreat() at first a reference of the CCMgr is fetched, as the bean is tightly coupled with the CCMgr. Then the method reconciliation4ThreatStarted() in the ReconciliationUtils class is called. It indicates, that reconciliation for this distinct threat is started. Now the reValidate() method of the CCManager returns the actual satisfaction degree of the constraint. There are three possibilities:

- POSSIBLY_SATISFIED, POSSIBLY_VIOLATED and UNCHECKABLE: The reconciliation is postponed until no more partitions exists.

- SATISFIED: The threat did not lead to a violation of the corresponding constraint and the constraint reconciliation process is not necessary.

- VIOLATED: The threat lead to a violation of the corresponding constraint and requires a state rollback.

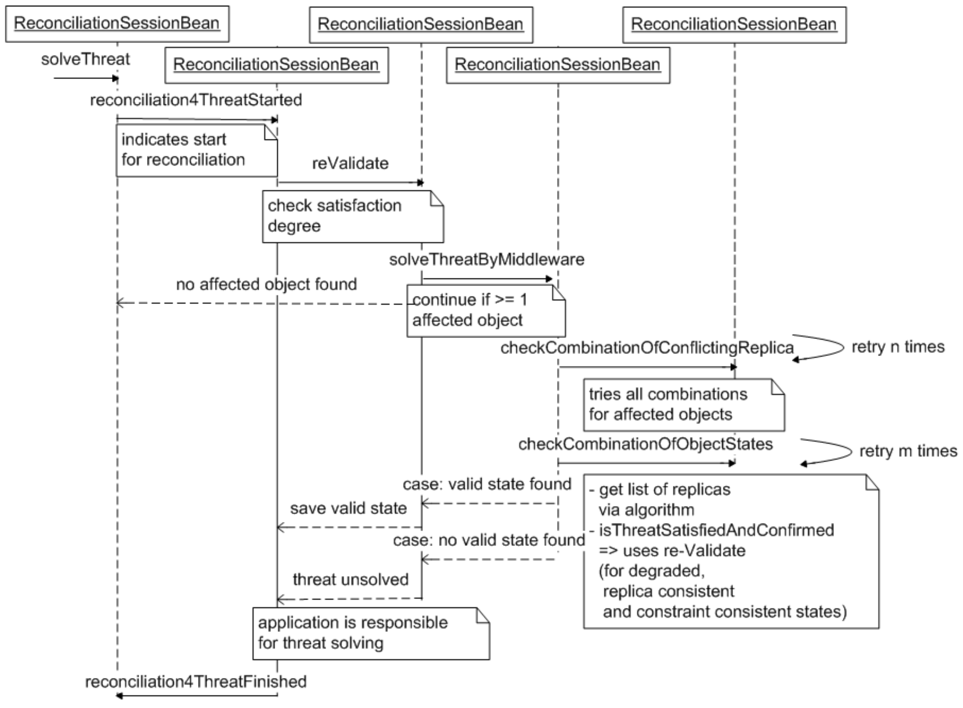

ReconciliationSessionBean
ReconciliationSessionBean
ReconciliationSessionBean
ReconciliationSessionBean
ReconciliationSessionBean
solveThreat
reconciliation4ThreatStarted
indicates start for reconciliation
reValidate
check satisfaction degree
solveThreatByMiddleware
no affected object found
continue if >= 1 affected object
checkCombinationOfConflictingReplica
retry n times
tries all combinations for affected objects
checkCombinationOfObjectStates
retry m times
case: valid state found
save valid state
case: no valid state found
threat unsolved
- get list of replicas via algorithm
- isThreatSatisfiedAndConfirmed => uses re-Validate (for degraded, replica consistent and constraint consistent states)
application is responsible for threat solving
reconciliation4ThreatFinished

Figure 13: Activity path for state rollback within the ReconciliationSessionBean

Afterward, the constraint priority is checked. The constraint reconciliation process continues, if it is a regular constraint or a relaxable constraints which can be treated as a regular constraint. Then the first attempt for threat solving starts. All affected objects of the threat are marked as being modified by the middleware. Now the method solveThreatByMiddleware() fetches the initial state of the threat and continues if the threat has at least one affected object. Within this method checkCombinationOfConflictingReplicas() is called. It goes through all possible combinations of conflicting replicas for every affected object of the threat. For every combination a re-evaluation of the constraint is performed. If the constraint is satisfied, it depends on the reconciliation hint of the application whether a callback to the application is performed or the found solution is immediately taken as final result. In case the application does not confirm the found solution, the middleware continues with the not yet proceeded combinations. The method is called recursively for all affected objects. If a solution is found, it is persisted by the middleware. That shall assure that threats that are created during middleware activities are persisted in order not to “forget” them and to process them later on. In the case that no solution is found, solveThreatByMiddleware() performs the next step. The method useOlderIdenticalThreatsForSolvingByMiddleware() of the class ReconciliationConfiguration returns if all identical threats may be used. Otherwise, only the youngest threat is allowed to be used. After this kind of check, the method checkCombinationsOfObjectStates() creates the data structure for the states which have to be validated. Now the history service comes into play: At first, the history states with the “degraded” tag are loaded into the state vector - preprocessed by the chosen algorithm like RoundRobin. The method isThreatSatisfiedAndConfirmedForVector() sets the affected objects of mainThreat to the ones in states, and if requested by the application, asks the application for its confirmation. It stores the current object state (via CCPersister) and tries out all states with a submethod: statesSatisfyThreat() sets a state of the vector to the component handle of the replica and uses the reValidate() method of the CCManager to get the actual satisfaction degree of the constraint. This is done until a solution is found with a state of the degraded data structure or all elements within this vector are tested. If no solution is found, the same procedure like the one for degraded states happens again within the checkCombinationsOfObjectStates() for replica-consistent and, as a last chance, with constraint-consistent history states. If a threat is satisfied, specific actions are taken by with the CCManager, like confirming that the threat is solved and information regarding the threat is deleted. The method call of solveThreatByMiddleware() is finished and solveThreat() continues the process. Another way to return to solveThreat() is that the reconciliation is impossible due to a missing constraint consistent state. This happens, for example, if someone cleaned the state history database. However, after that the threat is unlocked again with reconciliation4ThreatFinished(). In case a solution is found all affected objects of the threat are set constraint-consistent and a new constraint-consistent history entry for the state is created. If no solution is found, a flag within the threats' database entry is set and the external application is informed to be responsible for solving the threat. Of course, an object is set constraint-consistent, if there are no constraints on the object any more, which are threatened by the object.

**Data structure for state rollback.** The method checkCombinationsOfObjectStates() calls the history service for a map of all degraded history entries of all system nodes. There are several possible algorithms to build such a data structure. As part of this thesis' an interface IMapBuilder with the createMap() method is introduced. The following principles are implemented:

- With ConsecutiveNodes: The degraded history of node one (first entry in compmonitor-config.xml) is searched for a constraint-consistent solution, by starting with the lowest version of a replica going to the lowest version. If no constraint-consistent state is found the next node in the list is tried out, until all possible nodes have been searched through. This solution has the advantage of a fast solution, if one of the first nodes has a suitable state for a replica.

- With RevertedRoundRobin: Here the highest version of a replica is used. Alternating from the existing lists, it starts at the first node in the compmonitor-config.xml until it reaches the last node . This solution is of high quality, since the path through the different replica states first considers the the highest versions of all nodes, later trying the lower versions. A disadvantage is the possibly long duration cycle; if a solution is found when the constraint-braking takes place in the “degraded” mode early.

- With RoundRobin: This strategy differs from RevertedRoundRobin, as the lowest versions of a replica is tried out for consistency. Nevertheless an alternating way regarding the compmonitor-config.xml is also used. This solution shall provide a rather fast solution, as the replica states tried out are near the last consistent state. However, a clear disadvantage is the quality of the solution, as the probability that one of the first entries may fit is high. It is useful, if the consistency is broken with the first entry in the "degraded" mode and re-established in higher version due to e.g. deletion of a booking (if the flight booking application is used).

In Figure 14 the principle of each algorithm is shown. Old refers to history entries with a low version in the database, young refers to history entries with higher version. Thus these states are newer than the old ones.

See the evaluation in Section 5.2.2 for results regarding the usage of these principles within the state rollback process.

Additionally, the data structure OrderPreservingHashMap is used within the middleware. This map implementation uses an java.util.ArrayList to specify the order of the entries (keys) of the map. A java.util.HashMap is used for the key-value mapping.

**Postprocessing of constraint consistent states.** The aim of the DeDiSys middleware is to find the "best" solution. Another optimization goal is to re-establish consistency by a minimum number of rollbacks. This is difficult to reach, as the middleware comes with a generic solution path and cannot take into consideration the intentions of the designer of external applications. Thus, a history application prototype has been developed which allows the setting of states regarding the whole history of a replica, see Section 4.5 for further details.

## 4.5 History Application

Beside the implementations within the DeDiSys EJB framework, there are external applications which support different aspects of the middleware. One of them is the history application.

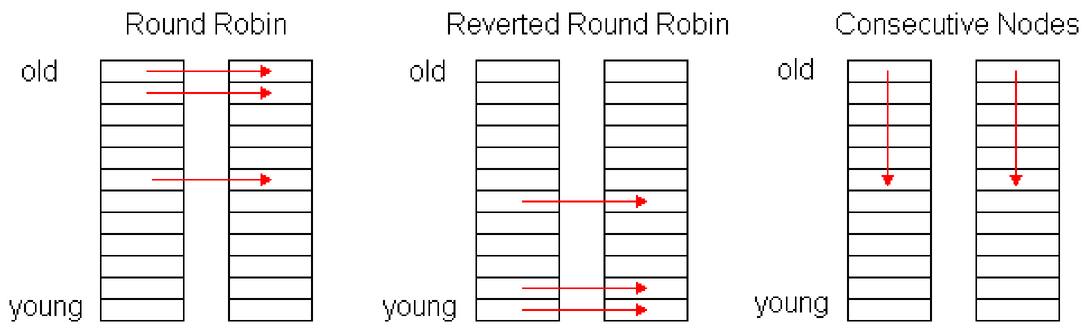


Figure 14: Implemented algorithms for the history data structure used in state rollback

### 4.5.1 Motivation for the History Application

The work carried out for this thesis shall improve the reconciliation phase when leaving the “degraded” mode to reach the “healthy” mode. One approach for this is the design of a sophisticated module for re-establishing a constraint-consistent system automatically. This requires not only a history database with a lot of stored information regarding the replicas, but also knowledge about the intentions of the user behind the applications. As a matter of fact, the DeDiSys goal is to find the “best” solution. The assumption is that the generic middleware solution-finding process does not tend to result in the “best” constraint-consistent state. Additionally, to the longer period of time the solution finding lasts, as the algorithms take time to be executed. On the other hand it is easy to provide an arbitrary “constraint-consistent” solution; for example, the last “constraint-consistent” state, which is stored in the history database anyway (if the history service is properly configured). Other simple solutions are declared in Section 4.4. When the system reaches the “constraint-consistent” state with such a valid, but not optimal, solution, the history application comes into play: A human operator might set better solutions for a distinct entity and optimizes the system state with the help of the history service. Technically, it is a roll-forward solution like the operation compensation described in Section 4.3.3 and performed with the help of an optimization tool which supports the reconciliation process.

### 4.5.2 Requirements for the History Application

The requirements for the history application listed here:

**Cooperation with history service.** The history application shall provide the functionality to create “constraint-consistent” states with the help of the history service. It shall be an external application, thus, not embedded within the middleware. The communication with the history service shall happen over J2EE prospects, like the session bean of the history service presented in Section 4.1.3. Main parts of the communication are the readout of history states and the propagation of new constraint consistent states.

**Graphical User Interface (GUI).** The human operator shall interact with the history application via a GUI. Hence, an easy-to-use GUI is required, which allows a fast selection and creation of replicas and history states. As this application is a prototype for applications where the user decides per mouse clicks about consistency, it is an essential criterion.

### 4.5.3 Functional Design of the History Application

In the following paragraphs, the functional design of the history application is provided:

**Description of the User Interface.** The history application has a main screen and a working screen, where constraint-consistent states can be set for the DeDiSys middleware. A screen shot is depicted as an example in Figure 15.

Generally, the user has to choose two different nodes as basis in the main screen. Afterward, he selects a replica element, which opens the working screen. Here the user makes a selection for a new constraint-consistent replica state and stores it in the database. Per default, the last consistent state is chosen.

The main screen includes the following elements:

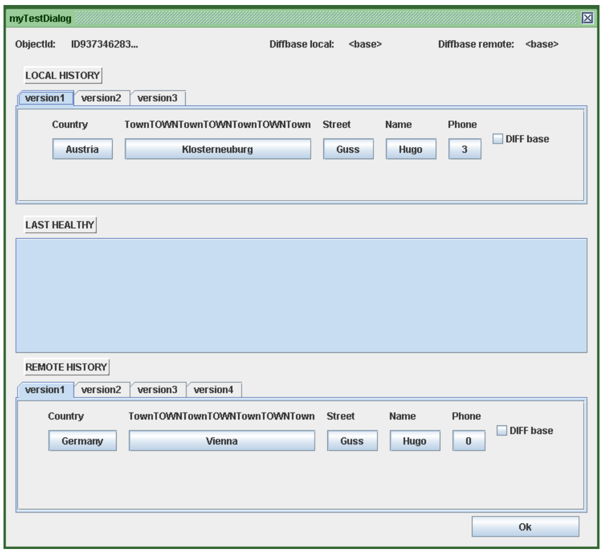
- List of available objects: In this list all replicas are listed which have at least one entry in the state history table. One replica object has to be chosen.
- List of available servers: This list shows all reachable nodes. Two server entries must be chosen for a distinct replica.
- Infobox: This is the message box of the history application for information for the user regarding events of the application.
- Button “DeleteHistoryDB”: Using this function deletes the complete history tables of states and operation entries. This function must be used carefully.
- Button “Update”: Whenever the button “Update” is pressed, the list of available objects and list of available servers is updated. For example, when a node joins a partition, an update is useful.
- Button “Quit”: This button ends the history application.

The working screen contains the following elements:

- Field “ObjectId”: This contains the object identifier of the replica.
- Tabbed Pane “last consistent”: This tabbed pane contains the clickable state attributes of the last consistent state of the replica.
- Tabbed Panes “Server one” and “Server two”: These two tabbed panes contain the clickable state attributes of all history entries ordered by version of the given replica.
- DIFF-checkmarks: Used for the DIFF-functionality.
- Buttons “OK” and “Cancel”: “OK” stores the chosen new constraint-consistent replica and distributes it to all nodes. “Cancel” discards all changes.

The working screen shows the attributes of the replica state. As a constraint, only simple types like string and integer are changeable. Complex data types are not changeable.

The history application provides a DIFF-functionality: DIFF means that the history application checks, if two attributes are identical. If one arbitrary state attribute of one



ObjectId: ID937346283... Diffbase local: <base> Diffbase remote: <base>

LOCAL HISTORY version1

| Country | TownTOWNTownTOWNTownTOWNTown | Street | Name | Phone |
|---|---|---|---|---|
| Austria | Klosterneuburg | Guss | Hugo | 3 |

REMOTE HISTORY version1

| Country | TownTOWNTownTOWNTownTOWNTown | Street | Name | Phone |
|---|---|---|---|---|
| Germany | Vienna | Guss | Hugo | 0 |

Figure 15: History application

of the two server nodes history or the last consistent state and the appropriate checkmark in the container is clicked this attribute serves as a basis for the same attributes within the other versions of this replica of the given node. If an attribute of the last consistent version is chosen, DIFF calculates all shown states. Three different colors are used to demonstrate a difference. “Red” indicates another value as the base. If the field is “yellow” it has the same value like the DIFF-base but at least one version between the base and “yellow” one has a different value for that attribute. “Grey” shows that the field entries are the same in the whole version stack from base to the “grey“-field. The history application with DIFF may not be confused with the ConfigurableReconciliationHandler, which is an alternative reconciliation handler presented in (Ertl and Ji, 2006), as the history of the replicas is not considered there.

**Swing for the GUI.** In Java there are two different options how to create a GUI: First, there exists the native Abstract Window Toolkit (AWT) which makes use of the operating systems components. But native components suffer from several deficiencies: They are not platform independent, so only a small number of elements are provided for all Java capable platforms. And they are not extensible. An alternative to AWT is the “Swing” library implemented purely in Java. In contrast to AWT, Swing has a worse performance in creating the graphics, as all elements have to be drawn with Java. Modern CPUs compensate this disadvantage. For further details see (Eckel, 2000). Within this project, the Swing library has been chosen due to the number of graphical components which are available.

The GUI of the history application has been created with the Integrated Development Environment (IDE) Netbeans, which has an internal GUI Builder named “Matisse”. With its “drag and drop” way of use for creating, resizing and adjusting of components, one can create a considerable user interface easily. Naturally, there are a number of components which have to be created programmatically, like the attribute labels of the replicas in the reconciliation window.

**Communication with the DeDiSys middleware.** An aspect regarding the business logic of the history application is the communication with the DeDiSys framework. During this work the standard EJB communication principles are used. For a better understanding they are depicted in the following example. The JNDI (see 2.1 and (Shannon, 2003)) acts as a guidepost. In Listing 11 the process is documented briefly: The history service (as an external application) needs a reference to the history session to further call the history service. The attributes of the Properties object are set according to the system configuration and an Initialcontext is generated. Then the lookup() method of the Initialcontext provides a HistoryHome object. With a Home object one can create a reference to the history service and call needed functionality like the state propagation or deletion of the whole history.

Listing 11: Communication between history application and DeDiSys framework

```
...
Properties p = new Properties();
p.put(``java.naming.factory.initial'',``org.jnp.interfaces.
    NamingContextFactory``);
p.put(``java.naming.provider.url'', ``jnp://localhost:1099``);
p.put(``java.naming.factory.url.pkgs'', ``org.jboss.naming:org.jnp.
    interfaces``);
InitialContext _ic = new InitialContext(p);
Object _o=_ic.lookup(``HistorySession``);
HistorySessionHome _historyHome=(HistorySessionHome)
    PortableRemoteObject.narrow(_o,HistorySessionHome.class);
...
HistorySession _history=_historyHome.create();
...
_history.deleteAllStatesHistory(replica);
```

## 4.6 Flight Booking Application

The flight booking application is a prototype for an external application which uses the DeDiSys middleware for constraint validation. During this project, several adaptations were needed for this type of application, which are documented here:

### 4.6.1 Motivation for the Flight Booking Application

The DeDiSys EJB framework is a middleware and a generic approach for constraint validation. Its functionality is the possibility to allow a trade-off between consistency and availability. To validate the theoretical approaches within the research project, an external application is needed. Thus, this application shall use the constraint consistency management. The flight booking application implemented and described in various thesis, for example (Baumgartner, 2007) is a suitable approach to these requirements. It comes with a Java client and a Webservice application. Further discussions about the client of the flight booking application concern only the Webservice one.

### 4.6.2 Functional Design of the Flight Booking Application

The flight booking server application is an EJB module which is deployed into the JBoss server. The main logic is implemented in the session bean flight bookingFacadeBean which manages the different entity beans FlightBean, PassengersBean, BookingBean and InvoiceBean, according to the method calls like addBooking(), deleteBooking(), addFlight() and so on.

With the client applications, one can use the functionality of the flight booking application. It is possible to create a flight with a unique identifier and add bookings to the flight. An arbitrary number of passengers can be added to the flight. As documented in Section 4.3.3, the total number of passengers must not exceed the defined value of seats in the economy or business class, otherwise a threat is created in the "degraded" mode. Additionally, bookings may be cancelled and flights be deleted.

As part of this thesis' work, the operation compensation mechanism for the method addBooking() has been implemented. See Section 4.3.3 for further details.

## 4.7 DeDiSys Control Application

A second external application for the DeDiSys middleware has been used and enhanced the DeDisys Control application.

### 4.7.1 Motivation for the DeDiSys Control Application

The DeDiSys EJB middleware is a comprehensive piece of software with several processes running (of course, processes within the middleware, not in the meaning of operating system processes). Thus, a steering application is useful. Such a tool would provide the possibility that a human operator can configure the system during its up-and-running - besides the configurations which are described in the different XML-files like compmonitor-config.xml. The requirements of such an application are dynamically driven by the test phases. Its main focus lies in the assistance of system evaluation to save implementation and testing time.

### 4.7.2 Functional Design of the DeDiSys Control Application

The DeDiSys control application consists of a main screen and a number of sub-screens. The implementation of this application has been done by several developers - the parts important for this thesis are highlighted below.

- JBoss: This is a text box where one can set the host for the functionalities illustrated below.

- Switch State: In this state, the system state can be set to “healthy” and a partition weight to a value between zero and one. This is used in case one node wants to simulate a distributed system.
- Clean replica list and history database: During a test session the cleaning of the replica list and history tables is necessary. Normally, the JBoss server is not restarted after a test case. The reason is the usage of the internal Java Virtual Machine and Just-In-Time (JIT) compilation, which needs one test run to deliver better timing measurements. The following tests require blank tables, adapted during this thesis’ work.
- Produce replica error: With the “switch state” command one might simulate a distributed system and thus needs replica errors for constraint and replica inconsistencies.
- Set algo for building state rollback map: As illustrated in Section 4.4.3, several algorithms for the state rollback map are usable. Thus, different test runs might be configured differently, depending on the algorithm (Implemented as part of this thesis).
- Set ReplicaReconciliationHandler: Within the compmonitor-config.xml file, a number of replica reconciliation handlers could be described. As only one reconciliation handler might be used for all entities, it is useful for different test cases to choose the one which fits best (Implemented as part of this thesis).
- Set MessageSize: The newly implemented history-based replication protocol has a configurable parameter - the message size. When a message is sent over the network with the group communication tool “Spread”, one can choose the maximum message size for such packets; also used to prove the performance enhancements of the new protocol (Implemented as part of this thesis).

The communication with the middleware is reached over the TestSessionBean and works like documented in Listing 11.

## 4.8 Implementation and Evaluation environment

The development and testing environment which has been used for this masters’ thesis are illustrated in the following. The weblinks to the technologies are provided in the appendix.

**Hardware.** The used hardware were standard personal computers (PC) with at least two GHZ processor speed and one Gigabyte RAM. The amount of hard disk space used for this project is negligible (less than one GB) on modern computers.

**Operating Systems.** The operating system for the development and testing was Microsoft Windows XP. Theoretically, it would work with Linux or derivatives, too, but this has not been tested. As the EJB environment requires Java, the Java Runtime Environment 5.0 represents the fundamental.

**Implementation environment.** For coding Netbeans 5.5 and Notepad++ were used. The compilation of the DeDiSys EJB project has been reached via "ant", a Java built tool. The DeDiSys EJB project is administrated with subversion (short: svn). For ease of use, svn has been used with the client application tortoise svn.

**DeDiSys framework.** The DeDiSys framework consists of a number of external and internal software packages, which form the deliverable:

- JBoss 4.0.4: This is the EJB container that is needed as a basis for the DeDiSys middleware.
- Spread: This is the group membership and group communication toolkit with release 3.17.4 . It has to be configured according to the used nodes with the node's IP address and an arbitrary name, which should be identical in all spread.conf - files. An example configuration can be found in Listing 12.
- MySQL: As persistence is an integral part of the thesis and the DeDiSys EJB environment, the MySQL server version 5.0 and MySQL tools for administration were needed.
- DeDiSys middleware: The EJB DeDiSys middleware is the intellectual property of the DeDiSys group and consists of a number of modules described in the section about fundamentals.
- DeDiSys applications: A number of applications have been developed on top of the EJB DeDiSys middleware. The most relevant for this thesis are the flight booking application, the DeDiSys test application, the control application and the history application described in the previous sections.

Listing 12: spread.conf

```
Spread_Segment 192.168.0.255:5050 {
        computer1               192.168.0.2
        computer2               192.168.0.3
}
```

**Software regarding the writing process of this thesis.** This thesis was written with LaTeX with the help of the editor environment TeXnicCenter.

# 5 Evaluation and Results

The DeDiSys project consists of a number of work packages. Amongst others, the work package 3.2 of the DeDiSys project required a stable version of the EJB middleware. For a correct delivery to the project inspectors of the European Union, a detailed test and evaluation run on the required system had to be made. This report can be found in (Künig (ed.), 2007).

A proof-of-concept shall be delivered with this thesis. Additionally, several quantitative measurements are provided to establish a relation to the existing solution. These measurements provide a basis for further discussions on the topic.

## 5.1 Test environment

The test environment used during the evaluation phase is described quickly:

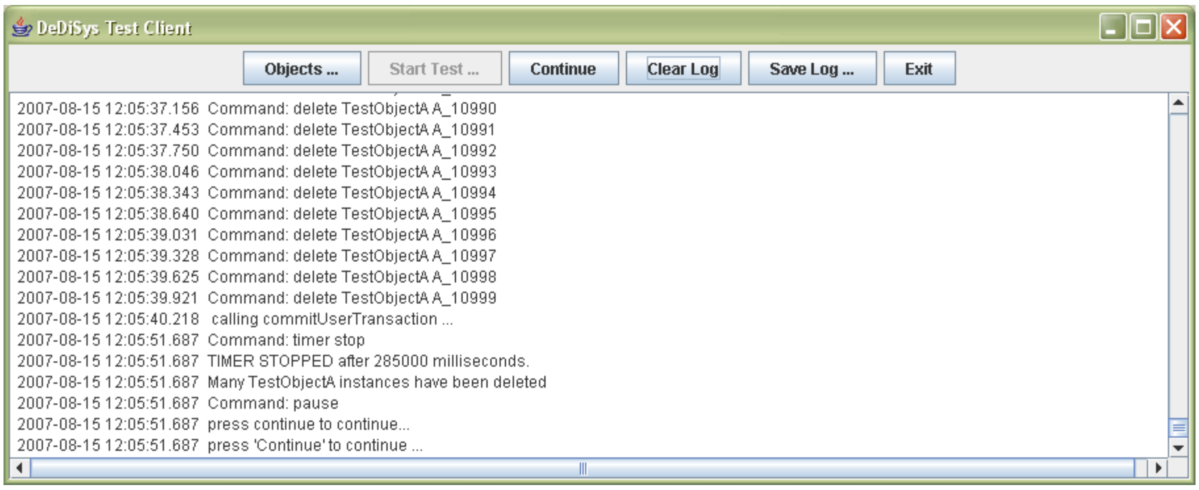
**System environment.** The testing environment is comparable to Section (Künig (ed.), 2007) and is described in detail in Section 4.8.

### Test applications

The DeDiSys-middleware functionality developed during this thesis is tested with two different applications, the DeDiSysTest application and the flight booking application.

**DeDiSysTest application.** As described in (Künig (ed.), 2007), the DeDiSys test application is an abstract EJB application with the sole purpose of showing and testing the capabilities of DeDiSys middleware without any further semantics. The application comes with a minimal user interface, see Figure 16 (an EJB client using Java swing). Its tasks are the interactive inspection and manipulation of its data model (two objects are available) by creating and deleting objects instances as well as displaying and setting attribute values. These actions on the test model (create, delete, get, set) and the invocation of operations can be triggered by the batch processing function. This allows the definition of complete test cases — consisting of huge numbers of steps — in XML format. Thus, complex repeatable test scenarios are supported. The application comes with several tests, but additional configurations for evaluations can be added. Hence, it acts as a well-engineered basis for different test scenarios as also performed for this thesis.

**Flight booking test application.** The flight booking test application is a client application which uses the flight booking beans (see Section 4.6) as a basis for quantitative measurements during the evaluation process. It was developed resulting from the needs of the different tests and might be adapted in the future according to the needs of the developers. Whereas the focus of the DeDiSys test application lies on the testing of basic



```
2007-08-15 12:05:37.156  Command: delete TestObjectA A_10990
2007-08-15 12:05:37.453  Command: delete TestObjectA A_10991
2007-08-15 12:05:37.750  Command: delete TestObjectA A_10992
2007-08-15 12:05:38.046  Command: delete TestObjectA A_10993
2007-08-15 12:05:38.343  Command: delete TestObjectA A_10994
2007-08-15 12:05:38.640  Command: delete TestObjectA A_10995
2007-08-15 12:05:39.031  Command: delete TestObjectA A_10996
2007-08-15 12:05:39.328  Command: delete TestObjectA A_10997
2007-08-15 12:05:39.625  Command: delete TestObjectA A_10998
2007-08-15 12:05:39.921  Command: delete TestObjectA A_10999
2007-08-15 12:05:40.218  calling commitUserTransaction ...
2007-08-15 12:05:51.687  Command: timer stop
2007-08-15 12:05:51.687  TIMER STOPPED after 285000 milliseconds.
2007-08-15 12:05:51.687  Many TestObjectA instances have been deleted
2007-08-15 12:05:51.687  Command: pause
2007-08-15 12:05:51.687  press continue to continue...
2007-08-15 12:05:51.687  press 'Continue' to continue ...
```

Figure 16: The DeDiSys test application

system functionality, the flight booking test application shall simulate the behavior of a system under more or less real conditions.

The Java client of the flight booking application has been enhanced to the same functionality as the flight booking application, such as adding of flights, bookings etc. Additionally, it allows to automatically create an arbitrary number of flight and booking objects. Such a functionality can be compared with the <loop> element of the DeDiSys test application. The principle is as follows: One creates a number of flights with the application. With the flights one can associate a number of bookings. A booking consists of several passengers. Each passenger counts for one seat within the flight — as the flight has a parameter "seats" which defines the total allowed passengers.

## 5.2 Test scenario: "Quantitative measurements"

During the implementation of the required features the following question arises: What are the differences in the performance of the two main constraint reconciliation strategies in the DeDiSys EJB environment — state rollback and operation compensation? The tests described in the following sections shall answer this issue. Additionally, a comparison of the different DeDiSys replication protocols is shown.

**Timing measurement.** As the quantitative measurements need time to compare system behavior, there are timers implemented in the DeDiSys middleware according to the test case, which provides reasonable values for the different test cycles.

**Statistical issues.** To average the results each of the following tests is repeated at least two times.

### 5.2.1 Test case: Replica Reconciliation comparison of the DeDiSys replication protocols

The pre-used DeDiSys replication protocol is compared with the new DeDiSys history based replication protocol concerning the duration of the replica reconciliation phase.

**Test specification.** This quantitative measurement shall provide a proportion in how far the new DeDiSys replication protocol speeds up the replication process after link failures are resolved. Thus, two subtests are necessary, one for each protocol.

The environment is set up as described in Section 4.8 with two systems. The system mode during this test is first in “healthy” mode and switches later into the “degraded” mode. Finally, the “healthy” mode shall be reached again. The state history is configured to “full” mode.

The DeDiSysTest application is used for these tests.

During the first “healthy” mode, different numbers of objects of type A, and according to the number of A, the same number of objects of type B, are created. Afterwards, a link failure will be accomplished which creates one partition with one node and another partition with two nodes. An update is done in each partition on all objects of type A and type B. Then the link between the two existing partitions will be reestablished and the replication phase with the DeDiSys replication protocol will occur.

The first subtests are made with six objects (three type A, three type B), ten objects, 20 objects, 40 objects, 60 objects, 200 objects and 2.000 objects. The parameter “messagesize” is only available in the DedisysHistoryBasedProtocol. It is set according to the object number in such a way that an adjustable number of objects can be transported in one message. For the DeDiSysProtocol the message parameter is not usable, thus each replica object is sent within one message.

The second subtests measures the update propagation of 60 objects with different message sizes from the DeDiSysHistoryBasedProtocol and for comparison with the DeDiSysProtocol.

Two timers are used:

- One timer measures the duration of the update propagation.
- The second timer measures the duration of the whole replica reconciliation process.

**Test results.** This test proves the functionality and the optimizations of the improved replication protocol. As expected, the DeDiSysHistoryBasedProtocol is faster then the DeDiSysProtocol. Figure 17 illustrates the costs per object regarding the replicated number of objects. For the DeDiSysProtocol the value stabilizes at about 400 milliseconds, with higher costs for a small number of objects. On the other hand, the DeDiSysHistoryBasedProtocol is configured with the parameter “messagesize” to guarantee the update propagation with maximal 30 objects per message over the network. The underlying group communication toolkit “Spread” allows a total size of about 120 kilobyte. This corresponds to the size of about 30 objects of the DeDiSys test application. Hence, for a higher number of objects, more messages are necessary. This increases the duration of the update propagation process, see Figure 17 for further details.

In Figure 18 the second sub-test is shown. The update propagation times are measured with different values for the "messagesize" parameter for 60 objects. As one can see, the higher the messagesize, the lower the propagation time. It levels off at about 5.000 milliseconds. For comparison reasons the protocol used before is tested too, which has an absolute value of about 22.000 milliseconds, which is higher than the lowest usable numbers for the new protocol.

**Test values.** Two tables are presented in this paragraph. First, the average update propagation and replica reconciliation measurements with the message size parameter for the DeDiSysHistoryBasedProtocol are illustrated. The tests for 60 objects are done once for the DeDiSysProtocol and five times for the new protocol with a changed message size. Second, the average costs per objects regarding the number of replicated objects are provided.

The abbreviation are as follows:

- DP UP: DeDiSysProtocol Update Propagation time in milliseconds
- DP RR: DeDiSysProtocol Replica Reconciliation time in milliseconds
- DHP UP: DeDiSysHistoryProtocol Update Propagation time in milliseconds
- DHP RR: DeDiSysHistoryProtocol Replica Reconciliation time in milliseconds
- MsgSize conf: configured maximal message size via the control application in bytes
- MsgSize total: reached maximal message size during replica reconciliation in bytes

#### Update propagation time for replication protocols

See Table 1 for further details.

#### Average costs for propagating objects

See Table 2 for further details.

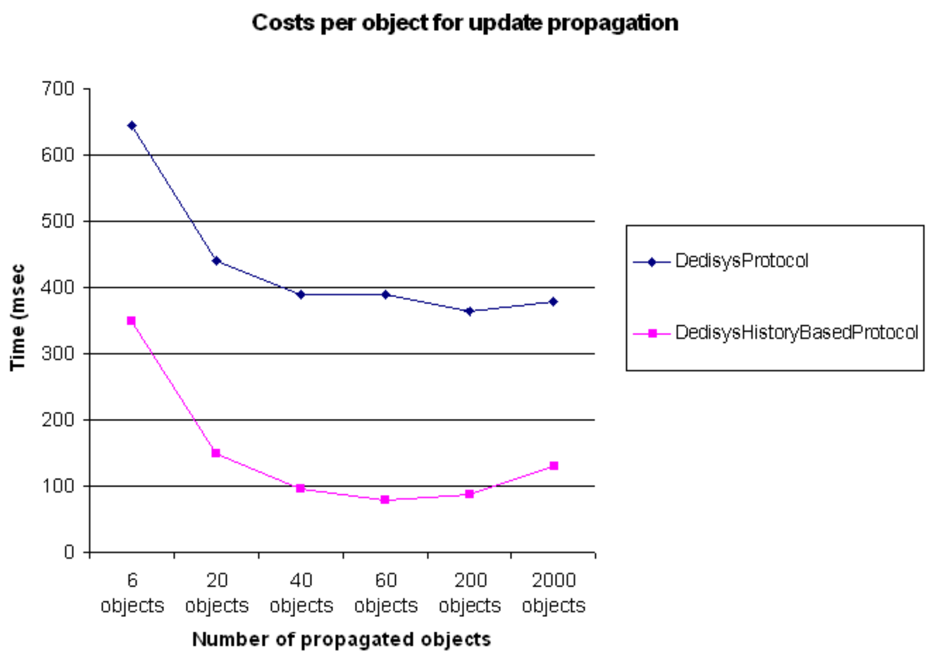


Figure 17: Costs per object for update propagation in milliseconds with specific message size

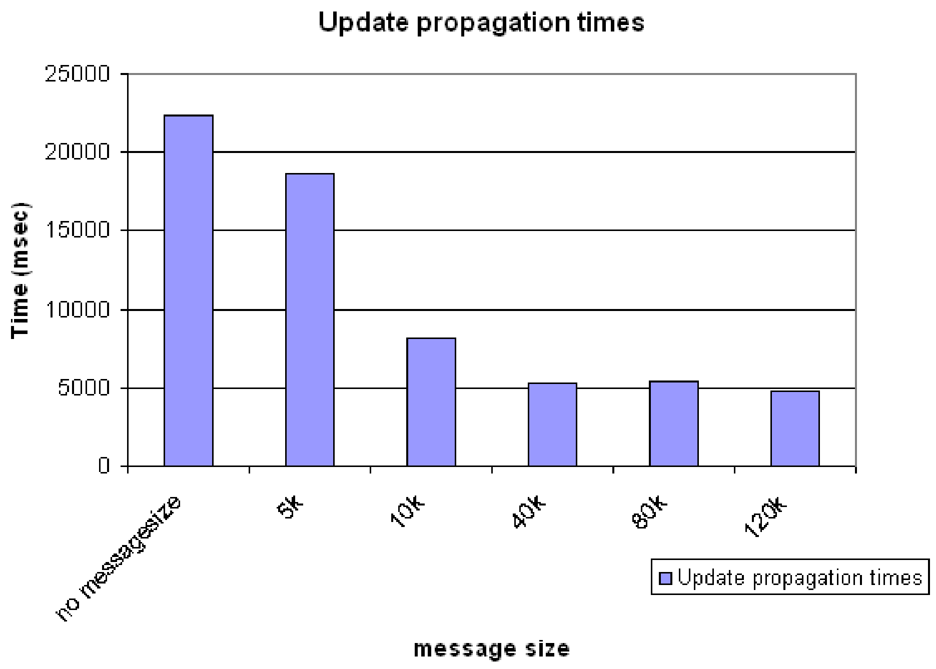


Figure 18: Duration of update propagation for 60 objects

| | DP UP | DP RR | DHP UP | DHP RR | MsgSize conf | Msgsize total |
|---|---|---|---|---|---|---|
| 6 objects | 3.866 | 3.881 | 1.048 | 1.048 | 10.000 | 12.726 |
| 20 objects | 8.750 | 8.782 | 2.967 | 3.015 | 40.000 | 37.337 |
| 40 objects | 15.555 | 15.656 | 3.813 | 3.813 | 80.000 | 70.378 |
| 60 objects | 22.351 | 22.492 | 4.735 | 4.735 | 120.000 | 103.3378 |
| 60 objects | | | 18.656 | 18.656 | 5.000 | |
| 60 objects | | | 8.164 | 8.172 | 10.000 | |
| 60 objects | | | 5.344 | 5.944 | 40.000 | |
| 60 objects | | | 5.453 | 8.485 | 80.000 | |
| 200 objects | 72625 | 72.633 | 17.547 | 18.450 | 120.000 | 122.695 |
| 2.000 objects | 757.938 | 757.968 | 259.032 | 259.093 | 120.000 | 123.028 |

Table 1: Update propagation time for replication protocols

| | 6 objects | 20 objects | 40 objects | 60 objects | 200 objects | 2.000 objects |
|---|---|---|---|---|---|---|
| DP | 644 | 440 | 389 | 390 | 363 | 378 |
| DHP | 350 | 148 | 95 | 79 | 87 | 129 |

Table 2: Average costs for propagating objects

### 5.2.2 Test case: State rollback - comparison of degraded history state list building

During the state rollback, a list of possible states has to be created, which is validated in the constraint consistency manager with one state after the other in the reconciliation phase. If one state fits the requirement of the state rollback (this means the threat is resolved and the constraint is satisfied) the state rollback for this threat is finished. There exist algorithms for building lists containing the states in a well-defined order, which allows a quick solution finding, whereas other algorithms provide better solutions than the fast ones in the given context. Therefore, three different algorithms for building a list of states are provided within the framework: “Round robin”, “reverted round robin” and “consecutive nodes”. The comparison shall provide an overview of how fast a solution can be found during state rollback. Information about the different algorithms is provided in Section 4.4.3. As a constraint the measuring of the quality of a solution is not possible without a given context. And the context is always application-specific.

**Test specification.** The environment is set up as described in Section 4.8 with two systems. The system mode during this test is first in “healthy” mode and switches later into the “degraded” mode. Finally, the “healthy” mode shall be reached again. The state history is configured to “full” mode.

The DeDiSysTest application is used for these tests.

During the first “healthy” mode one object of type A is created on the primary node and one object of type B is created on the secondary node. Afterward, a link failure is introduced. This leads to two partitions with one node each. Object A and B are updated 50 times in both partitions. Additionally, a threat is introduced to object A on the primary node. Updates occur as described in the following configurations:

- During the first configuration the constraint is broken with the first update in the degraded mode for object A. The following 49 updates continue the breaking of the constraint for object A.
- The second configuration leads to 24 updates of object A without breaking the constraint. But with the 25th update the constraint is broken. The following 25 updates still break the constraint.
- The third configuration updates object A 49 times without breaking the constraint. But with the 50th and last update the constraint is broken.
- The fourth configuration breakes no constraint during the 50 updates. However, the threat is introduced, too.

See Figure 19 for the described principles. Red indicates a constraint inconsistent state, green a constraint consistent state. "Old" refers to history entries with a low version in the database, "young" refers to history entries with higher version. Thus they are newer than the old ones.

As a threat is introduced for object A, the constraint reconciliation is triggered. Thus, the ReconciliationSessionBean attempts to solve the threat.

There is one timer used during each test. It measures the whole constraint reconciliation time for the solveThreat() method. As mentioned, the quality of a found solution is not measurable with the DeDiSys framework because different applications have different requirements.

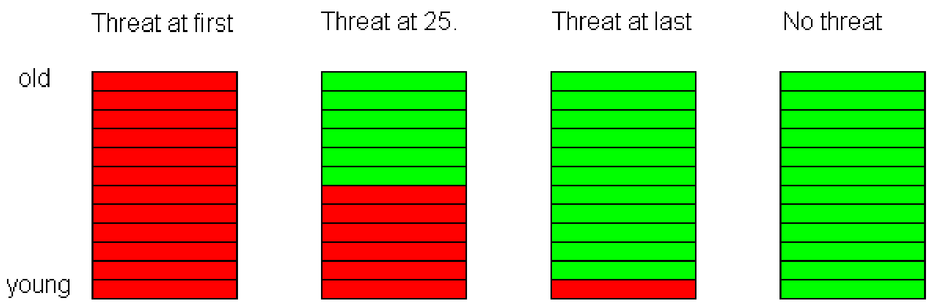


Figure 19: Used test cases for the comparison of different history data structure algorithms

**Test results.** With the processing of this test, it is proved that state rollback is fully functional. The usage of the history service for accessing the degraded states, replica-consistent states and last-consistent states deliver valid results. The three different algorithms build the data structure according to their specification. Consequently, the constraint reconciliation is working.

It was expected that the different algorithms lead to different execution times for constraint reconciliation within the same subtest. Interestingly, this is not the case. As one can see in Figure 20, they all provide nearly the same times regarding a specific test case.

The subtests among each other have different execution times:

- Threat at first object: This test has the longest execution time, nearly twice as long as “Threat at 25th object” and “Threat at last object”. It requires the loading of the complete history out of the state table. And all states attempt to solve the threat, but only the last constraint-consistent state is able to do so.
- Threat at 25th object: This test requires one database access as the threat can be solved with the degraded history.
- Threat at last object: Again, this test requires one database access as the threat can be solved with the degraded history.
- No threat: This test requires no database access, as the threat is solved with a comparison of the actual state against the expected state. Thus, it is the lower bound of a time period regarding the duration of the constraint reconciliation process.

These result show that the validation of a specific state does not last a long period of time. For example, “Threat at last object” with RevertedRoundRobin requires the process to validate about 100 objects (50 objects per partition). On the other hand, RoundRobin requires only two comparisons. Both algorithms provide nearly the same values; 9.990 milliseconds for RoundRobin against to 10.281 milliseconds for RevertedRoundRobin.

The DeDiSys CORBA and .NET middleware do not persist their objects. They hold them only in memory, which leads to faster processing times. As the DeDiSys EJB middleware uses persistence mechanisms, a subtest with “Threat at first object” is added, which measures the time for creating the data structure of rollback-states including the needed time for accessing the history database. Then all different states - degraded, replica-consistent and last-consistent - are needed. The result delivers averaged 187 milliseconds in contrast to the 15.593 milliseconds of the whole constraint reconciliation process (see Figure 21). Consequently, the persistence mechanism used for the history service is not the major burden in the reconciliation.

Therefore, further research is necessary to find out the time-consuming processes within the constraint reconciliation. This will help to improve the state rollback of the DeDiSys EJB framework.

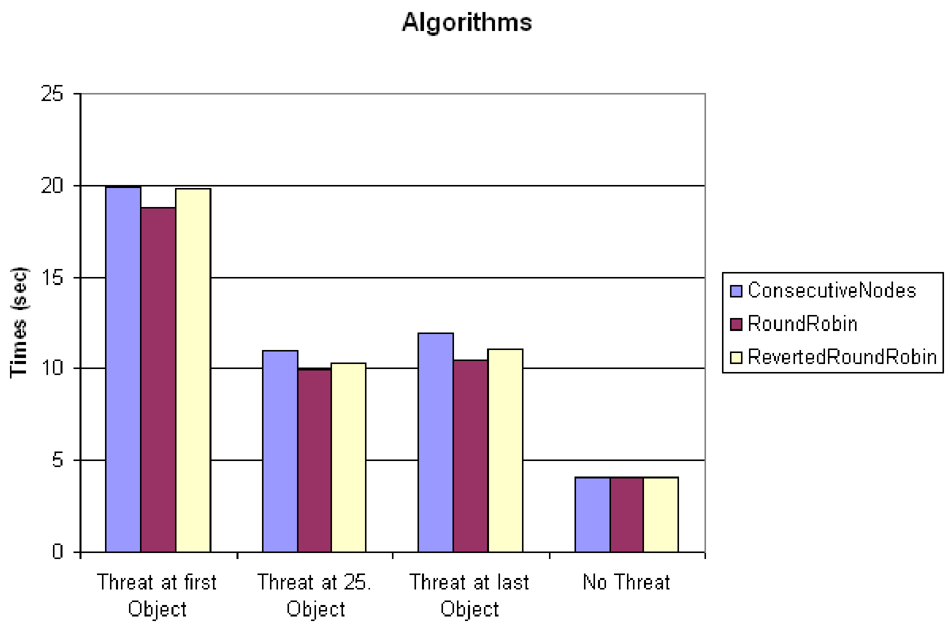


Figure 20: Reconciliation time with the different algorithms

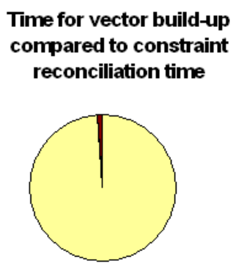


Figure 21: Time for history vector build-up compared to whole constraint reconciliation

**Test values.** The values for the different used algorithms for the constraint reconciliation process are given here. The timers for the measurements are placed in the class ReconciliationManager within the method reconciliationForApplication() from revision 303 of the DeDiSys EJB middleware. It starts at the very method begin, and stops with an output in the line before returning to the calling method. The values from the "No Threat" - evaluation have only to be tested once - as the constraint reconciliation makes not use of the history service as no threat has happened. Additionally, a table is provided which contains the measurements for accessing the history service database and creation of the data structure with the algorithms. The test shown here is the "Threat at first object" one with 50 updates with the consecutive node algorithm.

- Threat@1st: A threat is introduced with the first element.
- Threat@25th: A threat is introduced with the 25th element.
- Threat@last: A threat is introduced with the last (50th) element.
- No threat: No threat is introduced within this test.

### Consecutive nodes

See Table 3 for further details.

| Threat@1st | Threat@25th | Threat@last | No threat |
|---|---|---|---|
| 18.937 | 10.844 | 12.703 | 3.766 |
| 19.141 | 10.844 | 9.938 | 4.078 |
| 21.719 | 11.188 | 13.187 | 4.235 |
| 19.932 | 10.959 | 11.943 | 4.026 |

Table 3: Testing times for consecutive node test

### Round robin

See Table 4 for further details.

| Threat@1st | Threat@25th | Threat@last | No threat |
|---|---|---|---|
| 17.672 | 10.047 | 10.397 | 3.766 |
| 18.313 | 9.250 | 9.750 | 4.078 |
| 20.391 | 10.672 | 11.219 | 4.235 |
| 18.792 | 9.990 | 10.455 | 4.026 |

Table 4: Testing times for round robin test

### Reverted round robin

See Table 5 for further details.

| Threat@1st | Threat@25th | Threat@last | No threat |
|---|---|---|---|
| 17.625 | 10.015 | 8.672 | 3.766 |
| 20.782 | 11.687 | 10.484 | 4.078 |
| 21.109 | 8.750 | 14.047 | 4.235 |
| 19.839 | 10.281 | 11.068 | 4.026 |

Table 5: Testing times for reverted round robin test

### Times for history service access

See Table 6 for further details.

| Degraded data | Replica consistent data | Constraint consistent data | Total replication time |
|---|---|---|---|
| 47 | 0 | 15 | 15.593 |
| 47 | 15 | | |
| 46 | 16 | | |
| 140 | 31 | 15 | 15.593 |

Table 6: Times for accessing the history service

### 5.2.3 Test case: Comparison of operation compensation and state rollback

The DeDiSys framework shall provide an automatic rollback, either by an internal mechanism (state rollback) or via an external application (operation compensation). This test case compares these two implemented solutions: Whereas the state rollback keeps the whole reconciliation phase transparent to the external application (it can be seen as a generic solution strategy), the operation compensation allows an external application to provide its own solution strategies to create a new consistent state. This test case represents a fully-fledged DeDiSys system with the history service as reconciliation strategy supporter.

**Test specification.** The environment is set up as described in Section 4.8 with two systems. The system mode during this test is first in "healthy" mode and switches later into the "degraded" mode. Finally the "healthy" mode shall be reached again. The state history is configured to "full" mode.

The flight booking test application is used for these tests. During the first “healthy” mode, one or more flights (according to the subtest specification) with a total seat number of six are created. This seat number is the constraint which must not be exceeded during the “healthy” mode for booking passengers on the flight. Within the “degraded” mode an overbooking may occur that is detected during the following reconciliation phase. Thus, during the first “healthy” mode one booking with two or three passengers (see subtest specification below) is stored. In the following “degraded” mode one or more bookings on the flight happen equally on both nodes. Finally, replica and constraint reconciliation restore a valid state.

All subtests are done with both mechanisms, state rollback and operation compensation, to compare them within a fully-fledged system.

The subtest specifications are as follows. Each subtest is done with one flight, five flights and ten flights.

- 6=2/2+2: Within this test, two passengers are added in the “healthy” mode to the flights. In the “degraded” mode, two additional passengers are inserted on each node, leading to a total of six passengers. Thus, threats are introduced but the total number of allowed passengers with bookings is not exceeded.
- 6=3/3+3: Within this test three passengers are added in the “healthy” mode to the flights. In the “degraded” mode, three additional passengers are inserted on each node, leading to a total of nine passengers. Thus, threats are introduced and the total number of allowed passengers with bookings is exceeded.
- 6=3/9+9: Within this test three passengers are added in the “healthy” mode to the flights. In the “degraded” mode, nine additional passengers are inserted on each node, leading to a total of 21 passengers. Thus, threats are introduced and the total number of allowed passengers with bookings is exceeded.

A timer is integrated within the ReconciliationManager to measure the duration of the whole constraint reconciliation process. Other measurements are not considered within these tests.

**Test results.** With the execution of this test it is proven that state rollback is fully functional. During the processing of the operation compensation, the method cancelBooking() of the flight booking application throws an error on the backup node which is unable to delete distinct replicas. Neither the deletion of flights nor the cancellation of bookings is possible on backup nodes with the flight booking EJB as further investigations show. As the compensation mechanisms requires this functionality if a flight is overbooked, some results of the booked seats stored in the flight bean are not valid after operation compensation. Beside this bug, the operation compensation is working as expected.

As one can see in Figure 22 and 23, the costs per flight are nearly equal between operation compensation and state rollback. A “flight” consists of a flight entity bean, an invoice bean, a booking bean and several passenger beans. The subtest “6=2/2+2” has costs about five to six seconds, the subtest "6=3/3+3" about five to six seconds, too, and the subtest "6=3/9+9" average costs of 12 to 13.000 seconds. Interestingly, it does not matter if a flight is marginally overbooked (in case of "6=3/3+3") or not overbooked (in case of "6=2/2+2"). The threats produced for each booking within the degraded mode are the reason for this behavior, as the threat handling takes more time than the distinct threat resolving. And both test cases produce the same number of threats. Thus, the test case "6=3/9+9" has higher costs per flight as the threat number is increased.

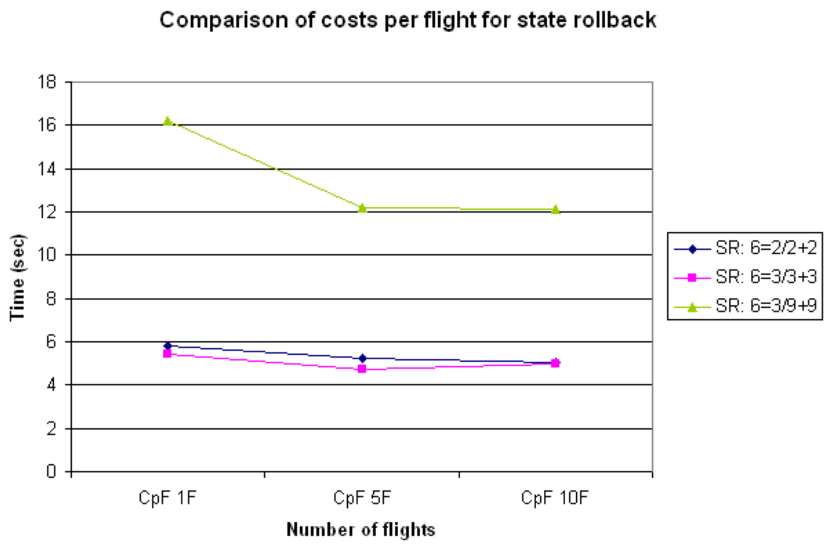


Figure 22: Costs per flight with state rollback

In Figure 24 one can see the times in seconds for the whole constraint reconciliation process. Operation compensation and state rollback times are both depicted. The overall results show what is indicated in Figures 22 and 23: Operation compensation as well as state rollback are comparable regarding their constraint reconciliation execution times. Generally, the more threats have to be solved, the longer the processing lasts.

Besides the threats which are desired to be solved, there are a number of threats which are not allowed to be rolled back by the system as they are not "REGULAR" ones. This adds additional time to the whole execution times.

**Test results.** The results for the different execution times for the constraint reconciliation mechanisms are presented here. The timers for the measurements are placed in the class ReconciliationManager within the method reconciliationForApplication() from revision 303 of the DeDiSys EJB middleware. It starts at the very method begin, and stops with an output in the line before returning to the calling method. State rollback is abbreviated with "SR" and operation compensation with "OC". "Cpf" means Cost per flight, "Avg" is average. The specifications for "6:2/2+2", "6:3/3+3" and "6:3/9+9" are described in Section 5.2.3.

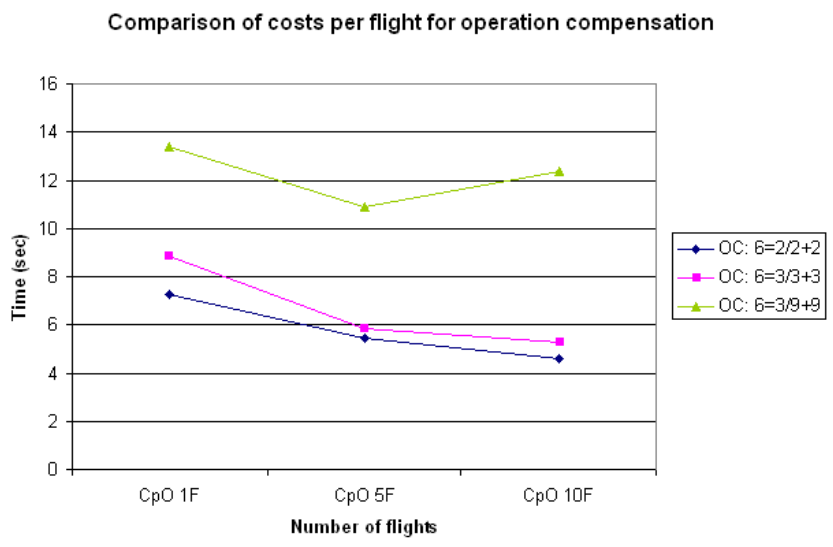


Figure 23: Costs per flight with operation compensation

## One flight

See Table 7 for further details.

| | SR\|6:2/2+2 | SR\|6:3/3+3 | SR\|6:3/9+9 | OC\|6:2/2+2 | OC\|6:3/3+3 | OC\|6:3/9+9 |
|---|---|---|---|---|---|---|
| | 6.187 | 4.907 | 16.266 | 7.578 | 9.765 | 13.563 |
| | 5.750 | 5.781 | 16.312 | 6.922 | 7.969 | 13.187 |
| | 5.562 | 5.531 | 15.969 | | | |
| Avg | 5.833 | 5.406 | 16.182 | 7.250 | 8.867 | 13.375 |
| CpF | 5.833 | 5.406 | 16.182 | 7.250 | 8.867 | 13.375 |

Table 7: Times for state rollback and operation compensation with one flight

## Five flights

See Table 8 for further details.

## Ten flights

See Table 9 for further details.

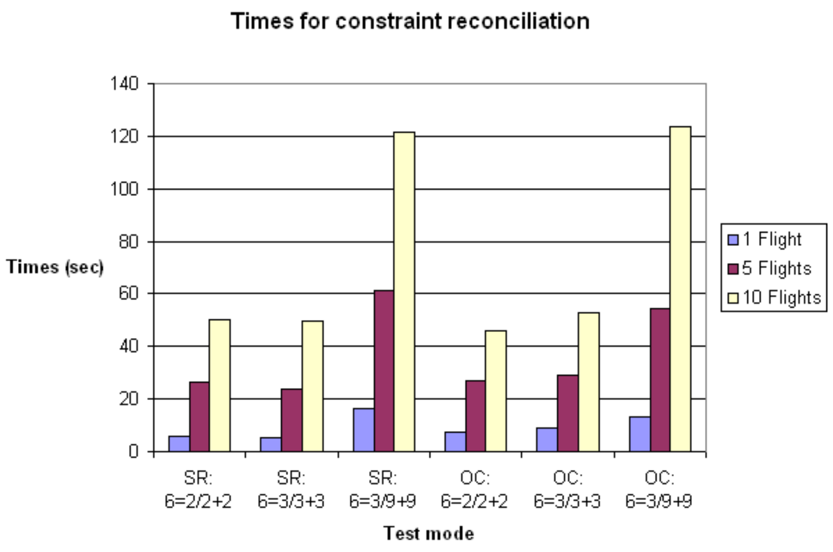


Figure 24: Times for constraint reconciliation with rollback and compensation

| | SR\|6:2/2+2 | SR\|6:3/3+3 | SR\|6:3/9+9 | OC\|6:2/2+2 | OC\|6:3/3+3 | OC\|6:3/9+9 |
|---|---|---|---|---|---|---|
| | 24.031<br>28.219<br>26.359 | 25.828<br>22.875<br>22.032 | 56.329<br>59.625<br>67.219 | 24.969<br>29.437 | 31.438<br>27.203 | 52.938<br>56.000 |
| Avg | 26.203 | 23.578 | 61.058 | 27.203 | 29.321 | 54.469 |
| CpF | 5.241 | 4.716 | 12.212 | 5.441 | 5.864 | 10.894 |

Table 8: Times for state rollback and operation compensation with five flights

| | SR\|6:2/2+2 | SR\|6:3/3+3 | SR\|6:3/9+9 | OC\|6:2/2+2 | OC\|6:3/3+3 | OC\|6:3/9+9 |
|---|---|---|---|---|---|---|
| | 50.532<br>53.859<br>46.256 | 51.141<br>48.313<br>49.500 | 119.781<br>123.360 | 48.032<br>43.687 | 57.343<br>48.023 | 125.625<br>121.406 |
| Avg | 50.216 | 49.651 | 121.571 | 45.860 | 52.773 | 123.516 |
| CpF | 5.022 | 4.965 | 12.157 | 4.586 | 5.277 | 12.352 |

Table 9: Times for state rollback and operation compensation with ten flights

# 6 Discussion

Within this section findings regarding the design and implementation of the contents are discussed. Unlike the detailed descriptions within the prior sections, a more general overview about the distinct modules shall be presented. Finally, future prospects are made as there is still room for improvements.

## 6.1 Discussion of the Findings

The findings are presented in detail.

**Design and implementation phase.** It requires time to get acquainted with and informed about the J2EE and EJB environment. Concepts for the design of small software projects are not able to be mapped one-to-one on software within J2EE. And JBoss, in which the DeDiSys EJB software is integrated, is a complex piece of software, too. Additionally, the previous work, done by M. Horehled, K. Fuchshofer, G. Aigner and L. Froihofer required some orientation. Consequently, at the beginning of the work process, the learning curve was steep.

The test and evaluation (as was the implementation) required an attentive and detailed plan. Unlike on single node systems, the testing of a distributed system is a complex and time consuming task, especially when debugging is necessary. Additionally, the focus of this thesis' work is on the reconciliation phase, which is the most complex system phase. Generally, the testing and debugging time was elaborative and time-consuming. On the other hand, this lead to a high gain in experience.

Finally, it was a satisfactory feeling to operate a fully-fledged DeDiSys EJB system. Fully-fledged means that it uses a history service configured for both full state and operation history. After the degraded mode, the system makes use of the newly implemented replication protocol for faster update propagation. Finally, the constraint reconciliation phase delivers a correct solution either for state rollback or operation compensation. And the history application may be used during the healthy mode to set a new state, if the automatically chosen one does not fulfill the user requirements.

**Replication protocol.** With the new replication protocol the implementation improvements are depicted easily: As the tests show, the update propagation is up to four times better than the previous solution. Thus, the expectations have been excelled.

Unfortunately, the implementation regarding the two components ReplicationManager and DedisysProtocol within the DeDiSys EJB environment is more affiliated as necessary. This comes from the number of developers on the modules (ReplicationManager and DedisysProtocol, an earlier implementation were topics of other diploma theses) and their different concepts and ideas: So it calls the method afterpostCreate() in the ReplicationManager propagateCreate() within the DedisysProtocol(), which vice-versa uses the method getUpdateSynchronziation() of the ReplicationManager to check if a replica is created. This example shall show that the terms ReplicationManager and DedisysHistoryBasedProtocol are not one-to-one portable to the terms within Figure 4.

An idea that came up during implementation was to integrate a piping mechanism in the methods replicaCreated(), replicaUpdated() and replicaDeleted() in the Replication-Manager to speed up the system whenever a replica is created, updated or deleted (as the primary node sends messages to the backup nodes). However, this approach has been abolished, as read and write access regarding replicas are spread over the whole system.

**History service.** The history service is a new important aspect within the middleware, as it centralizes the storage of operation calls and states. Thus, fast access to the service is necessary. Additionally, it has to be decoupled from the constraint consistency management or replication service, as it has to be used from external applications, too. Generally, along with decoupling comes a speed limitation (due to the "contract" between the decoupled partners). The usage of the Session Bean and/or MBean tries to absorb these negative aspects — as both concepts are developed for such cases. During the tests the "full" history was the first choice. "Partial" history seems to be useful for special cases, when there are too many updates or storage is limited. Ideally, the history entries are deleted after the constraint reconciliation phase (at least with the help of the human operator). Within the evaluation phase only the "full" history was used. "Partial" and "no" are tested for functionality but were not compared with "full".

**Operation compensation.** Operation compensation has the advantage that the external application developer can tailor its compensation mechanism to distinct cases and method calls. Thus, the detailed logic for compensation is placed in the external application. On the other hand, the price of tighter coupling of middleware and external application is high. The general rollback principle of the DeDiSys middleware gets minimized. Consequently, further research regarding the compensation time and effort of the DeDiSys EJB framework in comparison to state rollback is necessary.

**State rollback.** The state rollback approach of the middleware allows application designers to make use of a distributed system with high availability (due to the optimistic approach). The interesting aspect is the enhanced error transparency to the user application, as threats are handled internally. The implemented approach works faulty-free (regarding the used test cases). More efficient solutions are thinkable. Then a re-design of the constraint reconciliation module is necessary, as the process of reconciliation is not developed for high execution speed but functionality. Generic approaches to reconciliation are often difficult to construct, as they cannot take applications into account. Hence, the middleware should support the application with respect to the reconciliation task, but also allow application semantics to guide the process of reconciliation. In the DeDiSys project this application semantics is encapsulated within constraints. It turned out that state-based reconciliation is easier to handle (compared to operation-based reconciliation), as they require less information about the history of operating in the degraded mode, see (Froihofer (ed.), 2007) for further details.

## 6.2 Future Prospects

Although the aims of the thesis have been reached, a number of items are left for future work. As the DeDiSys EJB middleware is a prototype, different enhancements are useful for research reasons.

**Replication of session beans.** The existing DeDiSys EJB framework does not support the replication of session beans within the replication protocol, as the focus lies only on entity beans. For example, the map _replicaConflicts, a parameter within the DedisysHistoryBasedProtocol, does not contain all beans which have entries within the history service. To cover other data items of the J2EE EJB specification, the DeDiSys EJB prototype should be enhanced. The consistency threat administration is an entry point for such an implementation. There, top level invocations are assigned to consistency threats.

**Replication manager.** Although the interfaces of the replication manager within diverse DeDiSys deliverables are well-defined, it is a conglomerate of a range of functions within the EJB implementation. It actually handles functionality which should be integrated in the replication protocol or other, not defined communication modules. Future work may decouple this mechanisms to provide a clean solution, see (Froihofer (ed.), 2005a).

**Application specific replica reconciliation handler.** In the file compmonitor-config.xml one can set a distinct replica reconciliation handler. For example, the ReplicaReconciliationFlight acts as a sophisticated reconciliation handler for the flight booking application, but if another application is used it works like the simple primary wins. Unlike to the replica reconciliation handler, the constraint reconciliation handler is defined within the ccdefinitions.xml, which is an application specific file. The herein set constraint reconciliation handler overrules the default constraint reconciliation handler of the DeDiSys middleware. Future work may adapt the configuration of the replica reconciliation handler within the EJB environment to allow an application specific replica reconciliation handling strategy.

**Replica and constraint reconciliation phase.** The reconciliation process presented in this thesis is split in two phases — the replica reconciliation and the following constraint reconciliation. An approach to speed up the reconciliation process is the combination of both phases into one phase. Therein the interaction between constraint consistency and replica consistency must be reconsidered.

**State rollback: data structure building algorithms for state rollback.** The state rollback, the generic approach of the reconciliation protocol to reach constraint consistency, has three different variants for the build-up of a data structure like a list, in which all the different states of the degraded mode are integrated. The state rollback mechanism tries out the list elements one by one to validate the state against the application's needs. Thus, an intelligent ordering is necessary to provide a constraint consistent solution fast. As part of the work carried out for this thesis, the algorithms RoundRobin, RevertedRoundRobin and ConsecutiveNodes are implemented. They provide a rather simple approach for building such a list of states. Hence, they may be used for a wide number of applications.

To speed up the state rollback, it would be of interest, whether the application could steer the build-up mechanism (an approach where the middleware is accountable for the build-up). This could be reached by communicating with the external application, which has to implement a new interface like e.g. AlgorithmHints, with which the application sets parameters or chooses an according algorithm. Another option would be the decoupling of the build-up mechanism from the middleware to let the external application be responsible. The middleware delivers the degraded states in a well-defined order together with additional information like version and reachable nodes. Now the external application may sort the list according to its needs, as the application developer knows best about the typical usage of the application. Consequently, the disadvantage of such an approach is the coupling of application and middleware, which is not the focus of the DeDiSys project.

On the other hand, a generic state rollback needs a lot of information via dynamic or static configuration to provide the best solution for the user. As different applications and users have different intentions, a tighter coupling seems to be necessary.

**State Rollback: Improve state rollback during replica reconciliation.** As described above, the constraint consistency mechanism needs a data structure filled with degraded states to provide a solution (in addition to the replica-consistent and constraint-consistent states, which are the last chance during the constraint reconciliation process). Independent of the structure creation process (by the middleware or - as encouraged here - by the external application), the ReconciliationSessionBean might get a data structure with too many entries. When the degraded phase lasts long and many updates occur to a huge number of objects, the constraint reconciliation process is likely to take too much time. Consider the following example: 10.000 objects within a partition experience 100 updates, each during a length of one hour. The system consists of three nodes. The worst case, if full history is chosen in the compmonitor-config.xml file, leads to 10.000 (objects) x 100 (updates) x 3 (nodes) = 3.000.000 history entries, which have to be validated (assuming that all updates provide a constraint threat). It is obvious that the reconciliation process takes a long time, which is not the intention of the DeDiSys approach.

Therefore, the prototype shall be enhanced with a speed-up mode. If the number of history entries in the degraded state structure exceeds a distinct number (this may be defined in the compmonitor-config.xml), only e.g. every tenth state is validated for constraint consistency.

Consequently, this would lead to a faster constraint reconciliation phase.

**Operation compensation: Handling of application timeouts for negotiation and reconciliation handling.** As mentioned in (Fuchshofer, 2006) from a middleware point of view, no special attention has been given to possible timeouts (from application side) during negotiation and reconciliation. This may be troublesome during the operation compensation, as the middleware is waiting for a response from application side. It is hard to distinguish whether the operation compensation is still ongoing from client side or an error has occurred. Hence, the following mechanism shall be integrated to handle a possible faulty application: The application or the middleware defines a time period the operation compensation process from the application may last. If the total time is reached and no solution is presented from the application the reconciliation process either displays an according message to the user or, more sophisticated, tries out state rollback if in case the states are stored during the degraded mode.

**Operation compensation:** The implemented compensation class within the flight booking application has a limitation: It works only, if the flights which are allowed to be booked in are created within the healthy mode. Then the constraint "PartitionSensitive" is coupled with the addBooking() method. If the flight is created within the "degraded" mode "PartitionSensitive" is hooked in addFlight(). Hence, addFlight() and not addBooking() are stored within the operation history table. The bookings in degraded mode for the flight are not stored in the operation history table. Compensation for addFlight() is not part of this thesis projects, so bookings on such created flights cannot be resolved. In future implementations this behavior shall fixed.

**History application - complex data types.** The state of an entity (and therefore the state of a replica) consists of several items. The data types for these items may be an arbitrary simple (like e.g. integer, long or string) or complex (application-dependent objects) one. As the middleware approach is a generic one, complex data types of external applications are not considered for the design of the history application. This means that only attributes which are strings or integers are displayed in the working screen. The used external application, the flight booking application, does not make use of more complex elements within the state of a flight entity or passenger entity as it is a prototype, too. Although the history application is fully functional, the flight booking application may be enhanced and the attributes in the working screen for creation of a constraint-consistent state have to be non-simple data types. Thus, it is necessary to include a mechanism with which the different objects are recognized and displayed in the working screen. This can be implemented similar to the compensation classes, which are defined in an according XML-file. Within this file, the mapping of the complex object to its presentation in the history application has to be defined. Additionally, the history application may allow the adaptation of existing attribute values. At the moment, the user may choose between different history entries for a given attribute, but has no possibility to add another value. Both enhancements are needed, if the history application shall be realized within a real-world environment. A comprehensive administration of states requires such a functionality.

# 7 Conclusion

This section provides a summary and conclusion of the previous sections. Hence, the most relevant decisions concerning design which were necessary are pointed out.

This diploma thesis is part of the European research project DeDiSys and covers the implementation and evaluation of a replication protocol in an Enterprise JavaBeans environment.

As the aim of DeDiSys is the research of the trade-off between availability and consistency, a prototype implementation is needed which proves the theoretical theses and approaches. This thesis contributes to the EJB approach. The other frameworks of DeDiSys — one with .NET and another in CORBA — are not targeted. As this kind of software is a middleware, the DeDiSys EJB framework is based on the JBoss application server, which allows the usage of session beans and entity beans — the data items and business items within the J2EE environment.

There are two main parts in the EJB middleware: The constraint consistency management is introduced within the DeDiSys project. It is responsible for constraint validation, detection of threats, support of constraint reconciliation and the storing of information about constraint consistency threats. The second part is the replication service. It consists of the replication protocol (it manages update propagation between systems as well as the timing of these updates) and the replication manager (it provides functionality, which is used by the different possible replication protocols). This masters thesis improves the replication service.

The systems' cycle consists of three phases: The healthy phase, in which all nodes are connected, the degraded phase, in which network partitions occur and updates to a data item may produce a consistency threat as well as the reconciliation phase. During the reconciliation phase the different partitions have to be made consistent again. The previously implemented primary-per-partition replication protocol is the basis, on which the reconciliation phase of the protocol is enhanced. For this purpose, a configurable history service is introduced. It provides the storing of operations and states of entity bean's replicas during the degraded mode. This functionality is needed during the reconciliation phase as older object states are taken into consideration. Additionally, it allows the access of this data via an interface for the middleware or external applications.

Reconciliation within the DeDiSys EJB middleware makes use of object copies on the backup nodes. Therefore, replication is often used to provide fault tolerance for better availability in case of node and link failures.

The adaption of the P4 protocol regards the reconciliation phase, according to the focus of the research project. At first, the replica reconciliation phase is enhanced. Replica reconciliation synchronizes the different replicas in the previously existing partitions. After the nodes are connected together, the different replicas on one node are propagated to all the other nodes. The improved protocol comes with a solution, speeding up the update propagation. During the following constraint reconciliation phase, parts of a state rollback implementation approach are extended to a fully functioning solution. Additionally, a new operation compensation mechanism is introduced as an alternative to state rollback: Now an external application, like e.g. the flight booking application, may use the operation compensation instead of the generic state rollback to create a constraint consistent state again. State rollback and operation compensation use the history service interface for their functionality.

Beside the described middleware mechanisms, a prototype implementation for an external history application is applied: A human operator may use such an application to connect to the history service and set another state during the healthy state for a given entity bean. This enhances the generic approach of the middleware, for a fast and simple solution for a replica state is easily changeable afterward.

The work presented in this thesis allows a fully-fledged DeDiSys EJB middleware, regarding the reconciliation phase. Consequently, a comprehensive evaluation regarding the performance happened. The new replication protocol provides a faster missed-update propagation during the replica reconciliation phase. The operation compensation and state rollback had comparable times for constraint reconciliation according to the tests. Both mechanisms are usable to reach a constraint consistent healthy system mode again and provide the expected reasonable performance.

## List of links

This section provides links to websites which are relevant during this thesis. All this links have been checked for validity on August, 16th, 2007:

- Apache Ant: http://ant.apache.org
- DeDiSys: http://www.dedisys.org
- Hibernate: http://www.hibernate.org
- HSQLDB: http://www.hsqldb.org
- JBoss: http://labs.jboss.com
- LaTeX: http://www.dante.de
- MySQL: http://www.mysql.org
- Netbeans: http://www.netbeans.org
- Notepad++: http://notepad-plus.sourceforge.net/uk/site.htm
- Spread: http://www.spread.org
- Subversion: http://subversion.tigris.org
- Subversion Tortoise: http://tortoisesvn.tigris.org
- TeXnicCenter: http://www.texniccenter.org

## References

Asplund, M. and Nadjm-Tehrani, S. (2006). Post-partition reconciliation protocol for maintaining consistency. 24

Asplund, M. and Nadjm-Tehrani, S. (2007). Measuring availability in optimistic partition-tolerant systems with data-constraint. Washington, DC, USA. IEEE CS. 24

Babaoglu, O., Bartoli, A., Maverick, V., Patarin1, S., Vuckovic, J., , and Wu, H. (2004). A framework for prototyping j2ee replication algorithms. 11

Baumgartner, M. (2007). Adaptive constraint-validierung in einer verteilten enterprise javabeans umgebung. Master's thesis, University of applied sciences (UAS) 'FH Technikum Wien'. 58

Beyer, S., Bañuls, M., Galdámez, P., Osrael, J., and Muñoz, J. (2006). Increasing availability in a replicated partitionable distributed object system. Int'l Conf. on Dependable Systems and Networks (DSN'06). 3, 24

Beyer, S., Muñoz, F., Bañuls, M., Galdámez, P., and Osrael, J. (2005). Hybrid replication protocols. Technical Report D1.2.1, DeDiSys Consortium. 20, 38

Davidson, S. B., Garcia-Molina, H., and Skeen, D. (1985). Consistency in a partitioned network: a survey. *ACM Comput. Surv.*, 17(3):341–370. 12, 23

DeMichiel, L. (2003). Enterprise javabeanstm specification. Technical Report Version 2.1, Sun Microsystems, Inc. 6

DeMichiel, L. and Keith, M. (2006). Jsr 220: Enterprise javabeanstm - ejb core contracts and requirements. Technical Report Version 3.0, Sun Microsystems, Inc. 6

Eckel, B. (2000). *Thinking in Java*. Prentice Hall. 56

Ertl, D. and Ji, X. (2006). Replica reconciliation handler. Technical report, Vienna University of Technology, Distributed Systems Group. 18, 21, 41, 56

Froihofer, L., Osrael, J., and Goeschka, K. M. (2006). Trading integrity for availability by means of explicit runtime constraints. In *Proc. 30th Int. Computer Software and Applications Conference*, pages 14–17, Washington, DC, USA. IEEE CS. 2

Froihofer, L., Osrael, J., and Goeschka, K. M. (2007). Decoupling constraint validation from business activities to improve dependability in distributed object systems. In *Proc. 2nd Int. Conf. on Availability, Reliability and Security*, pages 443–450. IEEE CS. ii, 13, 14, 15

Froihofer (ed.), L. (2005a). FTNS system model. Technical Report D2.2.1, DeDiSys Consortium. `http://www.dedisys.org/`. 12, 13, 17, 18, 38, 78

Froihofer (ed.), L. (2005b). Trade-off: Availability - consistency. Technical Report D1.1.1, DeDiSys Consortium. `http://www.dedisys.org/`. ii, 14, 18

Froihofer (ed.), L. (2007). FTNS technology comparison. Technical Report D4.1.1, DeDiSys Consortium. `http://www.dedisys.org/`. 12, 19, 21, 22, 77

Fuchshofer, K. (2006). Negotiation and reconciliation of consistency threats for enterprise javabeans applications. Master's thesis, University of applied sciences (UAS) 'FH Technikum Wien'. 21, 43, 46, 47, 50, 80

Horehled, M. (2006). Integration of an ejb constraint consistency management framework into the jboss application server. Master's thesis, University of applied sciences (UAS) 'FH Technikum Wien'. 10, 31, 32, 34, 36, 46

JBoss-Inc. (2006). The jboss 4 application server guide. Technical report, JBoss, Inc. 8, 9

Künig (ed.), H. (2007). FTNS/EJB system design & first prototype & test report. Technical Report D3.2.2, DeDiSys Consortium. `http://www.dedisys.org/`. 11, 41, 61

Osrael, J., Froihofer, L., , and Goeschka, K. M. (2007a). Availability/consistency balancing replication model. Washington, DC, USA. IEEE CS. 23

Osrael, J., Froihofer, L., Chlaupek, N., , and Goeschka, K. M. (2007b). Availability and performance of the adaptive voting replication protocol. Washington, DC, USA. IEEE CS. 22

Osrael, J., Froihofer, L., Goeschka, K. M., Beyer, S., Galdámez, P., and Muñoz, F. (2006). A system architecture for enhanced availability of tightly coupled distributed systems. In *Proc. 1st Int. Conf. on Availability, Reliability and Security*, pages 400–407, Washington, DC, USA. IEEE CS. ii, 12, 15, 16, 17, 18

Ricciardi, A., Schiper, A., and Birman, K. (1993). Understanding partitions and the "non-partition" assumptions. Washington, DC, USA. IEEE Proc. of Fourth Workshop on Future Trends of Distributed Systems. 2

Roman, E., Sriganesh, R. P., and Brose, G. (2005). *Mastering Enterprise Java BeansTM, Third Edition.* Wiley Publishing, Inc. ii, 5, 6, 7, 8, 32

Shannon, B. (2003). Javatm 2 platform enterprise edition specification. Technical Report v1.4, Sun Microsystems, Inc. 11, 34, 57

Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., and Alonso, G. (2000). Understanding replication in databases and distributed systems. In *Proc. of 20th Int. Conf. on Distributed Computing Systems (ICDCS'2000)*, Taipei, Taiwan, R.O.C. IEEE CS. ii, 18, 19, 20, 23

Yu, H. and Vahdat, A. (2002). Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282. 24