FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Relaxed non-blocking Distributed Transactions for the eXtensible Virtual Shared Memory

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Andreas Brückl
Matrikelnummer 0626657

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn
Mitwirkung: Projektass. Dipl.-Ing. Stefan Craß

Wien, 6. Mai 2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Relaxed non-blocking Distributed Transactions for the eXtensible Virtual Shared Memory

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

by

Andreas Brückl
Registration Number 0626657

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn
Assistance: Projektass. Dipl.-Ing. Stefan Craß

Vienna, 6. Mai 2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Andreas Brückl
Heide 17, 2120 Obersdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Danksagung

Diese Diplomarbeit und mein Informatikstudium an der Technischen Universität Wien wären ohne Unterstützung durch gewisse Personen nicht möglich gewesen.

Ich möchte mich bei eva Kühn und Stefan Craß bedanken, die mich fortlaufend durch ihre konstruktiven Anregungen und Ratschläge bei dem Verfassen der Arbeit unterstützt haben. Mein Dank gilt auch allen Kollegen des XVSM Technical Boards, die mir in den regelmäßigen Meetings bei Problemstellungen weitergeholfen haben.

Des Weiteren bedanke ich mich bei Tobias Dönz für seine Unterstützung bei der Umsetzung in MozartSpaces.

Abstract

In the development of distributed applications, especially communication and coordination are complex tasks. Therefore developers are using middleware technologies which are hiding the complexity and providing enterprise features like transactions out-of-the box. Especially middlewares which are following the space-based computing (SBC) paradigm are often used for coordination tasks.

But although space-based middlewares are simplifying the development of distributed applications, most of them can not be used for complex applications because they do not provide an essential enterprise feature: distributed transactions. Other middlewares which already provide this feature often rely on old commit algorithms which are having known problems in certain scenarios.

Therefore this work provides a flexible and concurrent distributed transaction model which is aligned on the space-based computing paradigm. The new model supports long-lived transactions and provides a high degree of consistency. As a positive side effect the evaluation shows that in certain scenarios the new transaction model performs better than a standard local transaction system.

Kurzfassung

In der Entwicklung von verteilten Anwendungen sind speziell die Kommunikation sowie die Koordination komplexe Aufgaben. Daher verwenden Entwickler Middlewaretechnologien, welche die Komplexität verstecken und die benötigten Enterprise-Features, wie zum Beispiel Transaktionen, bereits "out of the box" mitbringen. Besonders Middleware-Produkte, die auf dem Paradigma eines gemeinsamen Datenraums basieren (space-based computing, SBC), werden oft für Koordinierungsaufgaben verwendet.

Obwohl die auf einem zentralen Speicher basierten Middlewareprodukte die Entwicklung von verteilten Anwendungen vereinfachen, können viele von ihnen nicht für komplexere Anwendungen verwendet werden, da sie ein wichtiges Enterprise-Feature nicht unterstützen: Verteilte Transaktionen. Andere Middlewareprodukte, welche dieses Feature bereits unterstützen, verwenden ältere Commit-Algorithmen, welche in bestimmten Fällen Fehler aufweisen.

Daher stellt diese Arbeit ein flexibles, nebenläufiges und verteiltes Transaktionsmodell vor, welches an dem Paradigma eines gemeinsamen Datenraums orientiert ist. Das neue Modell unterstützt langlebige Transaktionen und bietet einen hohen Grad an Konsistenz. Als positiver Nebeneffekt zeigt sich in der Evaluierung, dass in bestimmten Szenarien das neue Transaktionsmodell performanter ist als ein normales lokales Transaktionssystem.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

2PC	Two Phase Commit
3PC	Three Phase Commit
API	Application Programming Interface
CAP	Consistency, Availability, Partition tolerance
CAPI	Core API
CICS	Customer Information Control System
CNF	Conjunctive normal form
FIFO	First in, First out
IMS	Information Management System
IPL	InterBase Parallel Language
J2EE	Java 2, Enterprise Edition
JTA	Java Transaction API
LLT	Long Lived Transaction
MAPI	Monitoring API
MDBS	Multi database system
SAT	Satisfiability
SBC	space-based computing
TLA	The Temporal Logic of Action
WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Service Definition Language
XVSM	eXtensible Virtual Shared Memory

CHAPTER 1

Introduction

The number of devices with an attached network interface and Internet access increases rapidly. As a result the coordination effort among them increases exponentially. A common method for lowering the complexity of distributed applications is to use a middleware – a software abstraction layer between the operating system and the application. Using a middleware has several advantages:

- a general application programming interface is provided (API)
- the coordination complexity is encapsulated into the middleware
- enterprise features such as transactions are provided out-of-the box

Space-based middlewares are using a central data space for the coordination of the distributed processes and have an additional advantage: The processes need not know each other because they are all communicating with the space. Some space-based middlewares like JavaSpaces [2] also provide distributed transaction handling, which is an essential feature for enterprise applications. However, today‘s space-based middlewares are all using the blocking Two-Phase-Commit (2PC) [Gra78] protocol which has one major drawback: If the centralized transaction manager crashes, the whole application is blocked.

1.1 Motivation and goals

The primary task of the space-based middleware XVSM (eXtensible Virtual Shared Memory) [eK05b] is to coordinate the communication between several nodes in a distributed application. For this task XVSM uses a shared memory (called space) where all participants operate on.

However when the size and/or the functionality of distributed applications increases it may not be sufficient to have only one space for the coordination.

Sometimes it makes sense to split different functionality or business processes into different independently operating applications. An example would be to have two separate applications for the Sales department and for the Accounting department. There would be two distributed applications with two spaces. Although both applications work independently, they have to communicate in certain situations (e.g. when a sales order has been confirmed).

The concept of XVSM is based on a microkernel architecture where additional functionalities are plugged on top of it. The microkernel itself only provides local transactions but does not support transactions which span over multiple spaces. Such functionalities have to be implemented as extensions which are plugged on top of the microkernel.

The goal of this work is to implement an extension for the XVSM middleware which provides distributed transactions. Therefore a relaxed distributed transaction model has to be designed which fulfils the following requirements:

- Support of long lived transactions (LLT)
- Support of asynchronous transactions
- High concurrency and performance
- Relaxed handling in case of single node errors
- Deterministic behavior in all possible error cases
- Easy to use for application developers

This new transaction model will further increase the functionality of XVSM and it will allow XVSM to cope with more complex business requirements. Therefore this thesis introduces two new relaxed transaction models:

- **Extended Paxos Commit** is an extension to the existing Paxos Commit protocol [GL06]. It allows to split transactions into several sub transactions without losing the high consistency of Paxos Commit. Transactions can commit even though not all sub transactions have committed successfully.
- **REXX Transactions** are workflow-based transactions which have to be fully specified in advance. Transactions (workflows) are specified as combination of sub transactions (actions). The execution and coordination of the transaction is accomplished by the transaction manager and only the result is passed back to the application.

Both transaction models have to be integrated into MozartSpaces and evaluated in terms of usability and performance.

1.2 Thesis Structure

This thesis is organized as follows: Chapter 2 provides an overview of existing distributed commit protocols and summarizes their facts in a table. Afterwards chapter 3 describes the background of XVSM and its reference implementation MozartSpaces. Chapter 4 focuses on the two new distributed transaction models which have been designed in the course of this work. Afterwards chapter 5 gives an overview of their implementation in MozartSpaces. The evaluation (chapter 6) compares the new transaction models in terms of performance and usability. Chapter 7 shows up areas of possible future work regarding the provided transaction models and their implementation. Afterwards chapter 8 summarizes and closes this thesis.

CHAPTER 2

Related work

Distributed transactions are transactions which span over two or more hosts on a network. All hosts are using their own local transaction system to perform their part of the distributed transaction. In the commit phase the commit protocol is responsible to coordinate and synchronize all local transactions so that all perform the same operation (either commit or abort). In contrast to local transactions distributed transactions have to cope with more complex error cases like permanent and temporary network and host failures.

However the characterization of distributed transactions in terms of the ACID properties is analogous to local transactions:

- **atomicity** - Atomicity assures that either all operations of the transaction are executed or none of them are executed.
- **consistency** - If a system has been consistent before a transaction, then it will also be consistent afterwards regardless whether the transaction commits or aborts.
- **isolation** - Isolation prevents other transactions to see intermediate states of a transaction.
- **durability** - Durability guarantees that the data of a transaction are stored persistently after the transaction has been completed.

Distributed transactions can only be ACID compliant if the local transaction system of all participating hosts as well as the commit protocol are ACID compliant.

2.1 Constraints

This section describes two general constraints which apply to distributed systems: the CAP theorem as well as the FLP impossibility result.

2.1.1 The CAP Theorem

The CAP theorem [Bre10] dictates that a distributed storage system (like a database or a virtual shared memory) can only have two of the following characteristics:

- **Consistency** - All nodes of the system must see the same data at any point in time. This can only be assured by ACID compliant transaction systems.
- **High Availability** - At any point in time the system must be available and responsive so that every request eventually receives a response.
- **Partition tolerance** - The system is available even if the connection to some nodes or to a group of nodes is lost.

Figure 2.1 graphically shows the dependency of these characteristics.

Although the CAP theorem is dedicated to distributed systems, it can be used to classify distributed transaction protocols as well. Therefore we consider a transaction protocol as a distributed system which provides the service *commit* to client applications. In all of the discussed transaction models the client communicates with the transaction managers if it wants to create, commit or rollback a transaction. So we can derive that the availability of a transaction protocol depends on the availability of the transaction managers. With this regard consistency means that all instances of a transaction manager (if more than one exists) are consistent. Partition tolerance then focuses on the availability of the transaction protocol if the connection to one or more transaction managers is lost.

2.1.2 The FLP Impossibility result

Distributed transactions are using consensus algorithms to assure that all nodes come to the same decision - either *commit* or *abort*. Therefore distributed transaction algorithms are effected by the FLP impossibility result [FLP85]. This paper proofs that in a fully asynchronous system there does not exist a consensus algorithm which tolerates one or more crash failures.

Therefore all consensus-based algorithms must somehow introduce a kind of synchrony (e.g. timeouts).

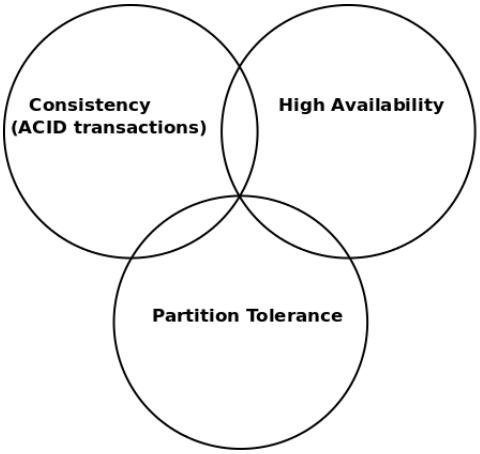

Figure 2.1: The CAP theorem: Dependency of the three characteristics

2.2 Distributed transaction models

2.2.1 2-Phase Commit

The Two-Phase-Commit (2PC) protocol [Gra78] [Lam81] is a distributed atomic commit protocol. Due to its long history and its low complexity related to other protocols, it is the most popular distributed commit protocol which follows the ACID properties. There exists one coordinator process which coordinates the protocol among all participants (cohorts). If the coordinator or any of the cohorts crashes, then the protocol blocks until all nodes have been recovered. Therefore it is called a blocking protocol.

As the name already says the 2PC consist of two phases:

- *prepare phase* - The coordinator process tells all cohorts to prepare the current transaction. Then all cohorts prepare their local transaction and acknowledge with either *Yes* or *abort*. In the first case the cohort from this moment on must assure to be able to perform both options: *commit* and *abort*. Since the protocol also covers node crashes, the cohorts have to write their status into a stable storage.

- *commit phase* - When all cohorts have replied in the first phase, the coordinator starts the second phase. It notifies all cohorts to either *commit* (if all have voted *Yes*) or *abort* (otherwise) the transaction.

Figure 2.2 shows the message flow of the protocol for the error-free case where *TM* is the transaction manager and *RM1* and *RM2* are the resource managers.

There are a lot of optimizations [LL93] [SBCM93] for the 2PC protocol which can be compared regarding the total costs of the protocol. The costs are split into three

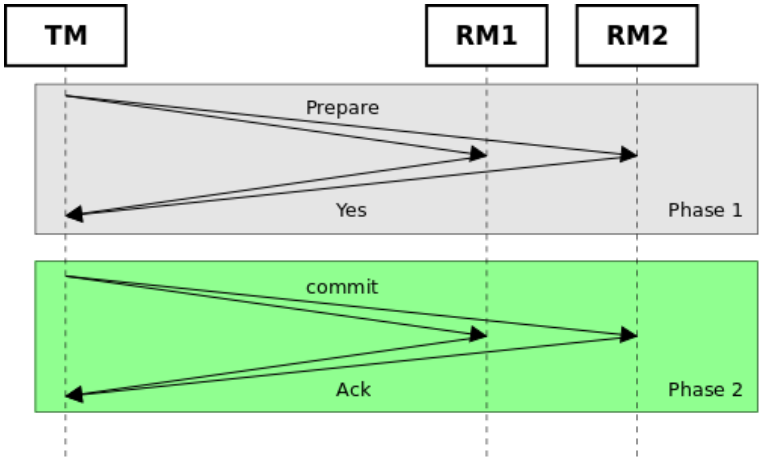

Figure 2.2: Sequence diagram of the 2-Phase Commit protocol

categories: number of messages, number of stable writes and message delay. All optimizations improve one or more of these categories.

The 2PC protocol is working properly in nearly all cases of system failures. However there are cases where the protocol gets stuck and has to be recovered manually by an operator. For example the protocol runs into an unrecoverable state if the coordinator decides to commit and tries to send the *commit* message to every cohort. The cohort then performs its local commit although it does not know whether the other cohorts have already received the *commit* message. If now the cohort and the coordinator crash and no other cohort has received the *commit* message then the protocol does not know the coordinators decision. Therefore it is not possible for a second coordinator to dependently recover and continue the protocol. If it would decide to commit, then there would be a consistency problem when the original decision has been *abort*. Otherwise if it would decide to abort and the crashed cohort has already committed, there would be an issue because the commit can not be undone.

Regarding to the CAP theorem (section 2.1.1) in the error-free case the 2PC protocol is consistent (since there is only one transaction manager) and available. But as soon as one cohort or the transaction manager crash it will become unavailable. Furthermore the protocol becomes inconsistent in the case mentioned above.
The 2PC protocol is for example used in the XA standard [5] for distributed transaction processing which has been developed by the The Open Group. The XA standard defines the communication between transaction managers and resource managers to execute distributed transaction. One implementation of the standard is the Java Transaction API (JTA) [9] which is part of the Java 2, Enterprise Edition (J2EE) . The space-based middleware GigaSpaces [10] uses JTA to participate in XA transactions. Also commercial middlewares like Oracle's Tuxedo [8] supports the XA interface.

Another architecture relying on the 2PC protocol is Jini [4] which is part of the Apache River project [3]. Jini is a network architecture which simplifies the development of distributed applications. It supports distributed transactions based on the 2PC protocol between several resources. Jini already comes with different transaction managers. The main difference to the XA standard is that Jini only manages transactions among Jini services whereas the XA standard also allows transactions among different kind of resources as long as they have implemented the XA interface.

JavaSpaces [FAH99] is a tuple-based coordination middleware which can use the Jini architecture to act as a service and so participate in distributed transactions.

IBM uses the 2PC protocol in their commercial middleware WebSphere MQ. Within a WebSphere MQ environment either CICS[1] or IMS[2] is used as the coordinator or transaction manager [1].

2.2.2 3-Phase Commit

The problem with the 2PC protocol is its blocking behavior in the event of certain system failures. Therefore the Three-Phase Commit protocol (3PC) [SS83] introduces timeouts as well as an additional phase to circumvent this problem. Here is a short description of the phases:

- *canCommit phase* - The coordinator asks all cohorts whether they can commit. All cohorts write their current status to the stable storage and reply with *Yes* or *No*. After a cohort has sent *Yes*, it has to be able to perform a commit even if it crashes in the meanwhile.

- *preCommit phase* - After the coordinator has collected all replies of the first phase, it either sends a *preCommit* message (if all cohorts have replied *Yes* within a certain time period) or an *abort* (otherwise) message to all cohorts. All cohorts then reply with *Ack*.

- *commit phase* - The coordinator will send an *abort* message to the cohorts if it has received at least one *abort* message or if it timed out when it was waiting for an answer. Otherwise, if the coordinator has received *Yes* from all cohorts it sends the final *commit* message.

Figure 2.3 shows the message flow of the protocol for the error-free case.

In contrary to the 2PC protocol the 3PC protocol ensures that a second coordinator can dependably take over after the primary coordinator crashed. The new coordinator has to query the state of all cohorts. If all have received the *preCommit* message the

[1] Customer Information Control System (CICS) is a transaction server developed by IBM.
[2] Information Management System (IMS) is a database which acts also as transaction manager.

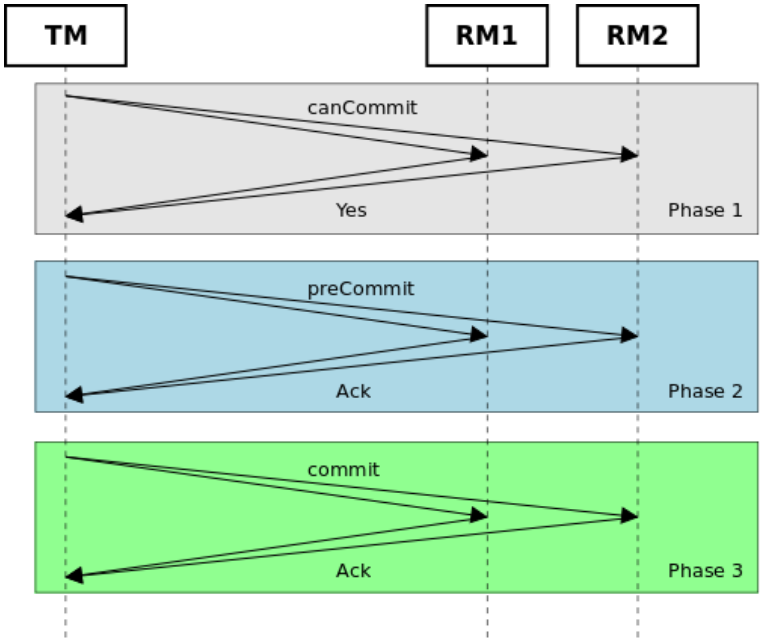

Figure 2.3: Sequence diagram of the 3-Phase Commit protocol

coordinator knows that the originally decision was *commit* and can resend the *commit* message. Otherwise it aborts the transaction by sending *abort*. Even if all coordinators crash, the timeout ensures that all transactions complete within a specified time. Therefore 3PC is also called a non-blocking protocol.

However, the 3PC protocol does not work correctly in all cases. Imagine the coordinator fails and the network segments into two subnets, where all cohorts of the first subnet have received the *preCommit* message and those of the second subnet have not. After querying all cohorts the new coordinator in the first subnet will decide *commit*. The new coordinator in the second subnet also queries all reachable cohorts but since none of them have received a *preCommit* message it decides to *abort*. When the two subnets will be merged later then there will be an inconsistent state because some of the cohorts have committed whereas the others have aborted the transaction.

Another drawback is the higher message overhead compared to the 2PC due to the additional phase. Maybe this is the reason why the 3PC has rarely been implemented. There is no established middleware or application server which has implemented 3PC. However 3PC has been used in a resource broker and reservation system [HDDG04].

2.2.3 Paxos Commit

The 2PC and 3PC protocol both rely on a single transaction manager which represents the single point of failure. If the transaction manager crashes the protocol is blocking.

Although the 3PC allows a secondary transaction manager to take over, there will be a lack of availability until it can continue operation. First the new transaction manager has to detect the crash of the primary transaction manager and afterwards all open transactions have to be processed according to their current state. The Paxos Commit algorithm [GL06] is based on the Paxos consensus algorithm [Lam98]. It uses several (*2f+1*) transaction managers (called acceptors) and can therefore tolerate up to f failures. As long as more than f acceptors are available the protocol operates without interruption.

Due to the higher complexity Paxos Commit requires more roles than 2PC and 3PC for its execution. Here is a list of all roles:

- *Resource Manager (RM)* - The resource manager is the interface between the local transaction manager of a cohort and the Paxos protocol. Every cohort on which transactional operations should be performed has to run an instance of the resource manager.

- *Registrar* - The registrar process is responsible to communicate the set of participants of the particular distributed transaction to the Paxos protocol. Every cohort which wants to take part in a transaction has to send a registration message to the registrar. A cohort is not allowed to perform a transactional operation before it has received the acknowledgement of the registration from the registrar. The registrar is not part of the original Paxos consensus algorithm because in the original algorithm only the final result is important and not the set of participating resource managers[3]. However in the commit algorithm the set of participants is part of the final result.

- *Acceptor* - The acceptor acts as the transaction manager of Paxos Commit. It collects the *Phase2a*-messages (*Prepared2a* or *Aborted2a*) of all participating cohorts as well as the *Phase2a*-message of the registrar (which contains the set of participants) and sends an aggregated *Phase2b*-message to the leader. This *Phase2b*-message is either *Prepared2b* (if only *Prepared2a* messages have been received) or *Aborted2b* (otherwise). The *Phase2b*-message also contains the set of participants. To tolerate up to f failures of acceptor nodes, the protocol requires at least *2f+1* acceptor nodes. An increase of the acceptor nodes also linearly increases the message complexity of the protocol. Since always a majority of acceptors is involved in the decision the consistency of the Paxos Commit protocol is guaranteed. If no majority is available the protocol blocks.

- *Leader* - Each transaction has one initial leader which guarantees liveness of the protocol. If the current leader crashes, then an election algorithm determines the next leader.

[3] In the original consensus algorithm the resource managers are called *learners*

In practice it makes sense to run several roles on the same node. A failure of the leader, the registrar or a resource manager does not effect the protocol since timeouts at the resource managers and at the acceptors assure that the transaction will be aborted.

Figure 2.4 shows an error-free execution of the Paxos Commit protocol with three acceptor nodes. The figure only shows the commit cycle. Here it is assumed that resource managers RM1 and RM2 have already sent their registrations to the registrar before.

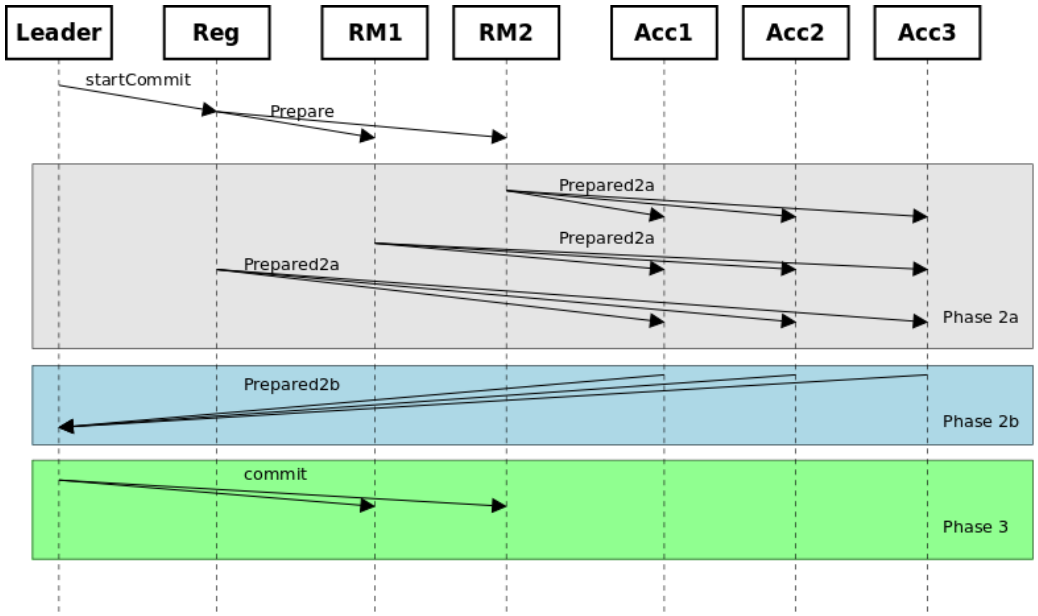

Figure 2.4: Sequence diagram of the Paxos Commit protocol

The commit cycle is started when the initial leader sends *startCommit* to the registrar process. Then the registrar closes the transaction for further participants and does not accept further registration messages. Afterwards it sends *prepare* to all resource managers which then prepare their local transactions. Afterwards the registrar and all resource managers communicate their commit decision (*prepared* or *abort*). Therefore each of them starts a new instance of the consensus algorithm and sends a *Phase2a*-message to all acceptors. If a cohort crashes before sending the *Phase2a*-message to a majority of acceptors the consensus protocol guarantees to abort the transaction. Per definition the registrar always sends a *Prepared2a*-message in its second phase which includes the correct set of participants.

For a successful execution of the protocol it is required that all resource managers as well as the registrar have voted *prepared* in their Paxos instances. If at least one instance has decided *abort* then the transaction will be aborted.

Before a resource manager is sending a *Prepared* message it has to store its decision as well as all other required data to a stable storage. Once a resource manager has sent its decision it must not revoke it. So in case it has communicated *Prepared* to the acceptor processes it commits itself to be able to handle a *Commit* as well as an *Abort* decision. Even if a resource manager crashes after sending *Prepared* it has to recover all required information.
In contrast to 2PC and 3PC Paxos Commit is always consistent, even in the case of network partitioning. However the availability is not guaranteed in that case. Availability is guaranteed if at least a majority of acceptors can be reached, otherwise the protocol blocks. So Paxos Commit tolerates a small degree of partitioning as long as a majority of acceptors is available for client applications. Paxos Commit always guarantees consistency, however it can not guarantee availability and partition tolerance at the same time (see CAP theorem in section 2.1.1).

The advantages compared to 2PC and 3PC are causing an additional communication overhead for every additional acceptor. In terms of message delays, number of messages and stable storage writes 2PC is a special case of Paxos Commit with a single acceptor [GL06].
The original Paxos consensus algorithm is used in the coordination service DepSpace [BACF08]. DepSpace provides a tuple space abstraction which implements replication for fault tolerance and access control for security. The implementation of replication is based on the Paxos algorithm. The data management middleware Sprint [CPW07] also uses the original Paxos consensus algorithm to orchestrate several in-memory databases running in a cluster. For the evaluation of the performance of the used total-order multicast approach, Sprint has also been implemented with Paxos Commit. The evaluation shows that in the assumed scenario the performance of the total-order multicast approach is better than the Paxos Commit implementation [CPW07].

2.2.4 Sagas

Already in 1987 Hector Garcia-Molina and Kenneth Salem described in their publication [GMS87] a method to handle long lived transactions (LLT). The problem of long lived transactions is that during their execution all of their objects are locked and not accessible by other transactions. This leads to bad performance and concurrency. Also the deadlock frequency grows with the fourth power of the transaction size [GMS87]. Due to their duration LLTs have a higher *abort* rate because the probability of a system crash of one of the nodes increases with the duration.

In the definition of the paper a Saga is a LLT which can be written as a sequence of transactions that can be interleaved with other transactions. A Saga consists of several sub transactions which commit immediately when they are completed. If the transaction manager later decides to abort, then the work of already committed sub transactions has to be undone. A Saga therefore stores for each sub transaction a compensation action.

Although the whole Saga is not executed atomically, its sub transactions are. Due to this relaxed isolation other transactions can access intermediate results of a Saga. If a Saga has been committed, then all of its sub transactions must have committed. Otherwise, if a Saga has been aborted, then all sub transactions have to be aborted or compensated.

A normal transaction manager provides at least the following commands:

- `startTransaction`
- `commitTransaction`
- `rollbackTransaction`

For developers the Saga model provides the following commands:

- `beginSaga` - Starts a new Saga.
- `beginTransaction` - Starts a new sub transaction within the Saga.
- `endTransaction` - This command commits the current transaction. This command also contains a reference to the related compensation action.
- `abortTransaction` - Aborts the current transaction but does not abort the surrounding Saga.
- `endSaga` - Commits the currently executing sub transaction if it has not yet been committed by an `endTransaction` and completes the Saga. There are no additional actions required since all sub transactions have already been committed or aborted.
- `abortSaga` - Aborts the current running transaction as well as the whole Saga. For all already committed sub transactions the related compensation action will be executed.

Sagas additionally provide the option to specify save points between transactions. Then in case of a system crash only transactions after the last save point need to be compensated. Afterwards the Saga can be restarted at the save point. This prevents the compensation of the whole Saga after a system crash and therefore decreases the compensation effort and prevents the re-execution of the transactions before the save point.

Transaction managers of database systems are using write-ahead logs to rollback a transaction after a system crash. But for a Saga this type of rollback is not applicable because sub transactions might have already committed and a compensation action would be required to undo the committed work. Therefore the transaction manager[4] needs to have access to the code of the compensation action even after a crash of the application.

[4]The transaction manager in the saga model is called saga execution component (SEC).

In the normal case the applications sequentially execute the transactions of a Saga by using the described commands. However it is possible to execute sub transactions in parallel if the application spawns new processes. Therefore the command `beginSaga` returns a reference which is used to start new transactions within an existing Saga. The model itself does not execute sub transactions in parallel.

The model assumes that the compensation action never fails. The developer must assure that the compensation action always completes successfully, otherwise the system will become inconsistent. A Saga is only deployed on a single node whereas the discussed ACID compliant protocols (2PC, 3PC, Paxos Commit) require protocol agents (resource managers) on every cohort. Also no distributed commit protocol is required, since the transaction manager directly passes all SQL (Structured Query Language) commands to the corresponding remote database managers which are acting in this model as resource managers.

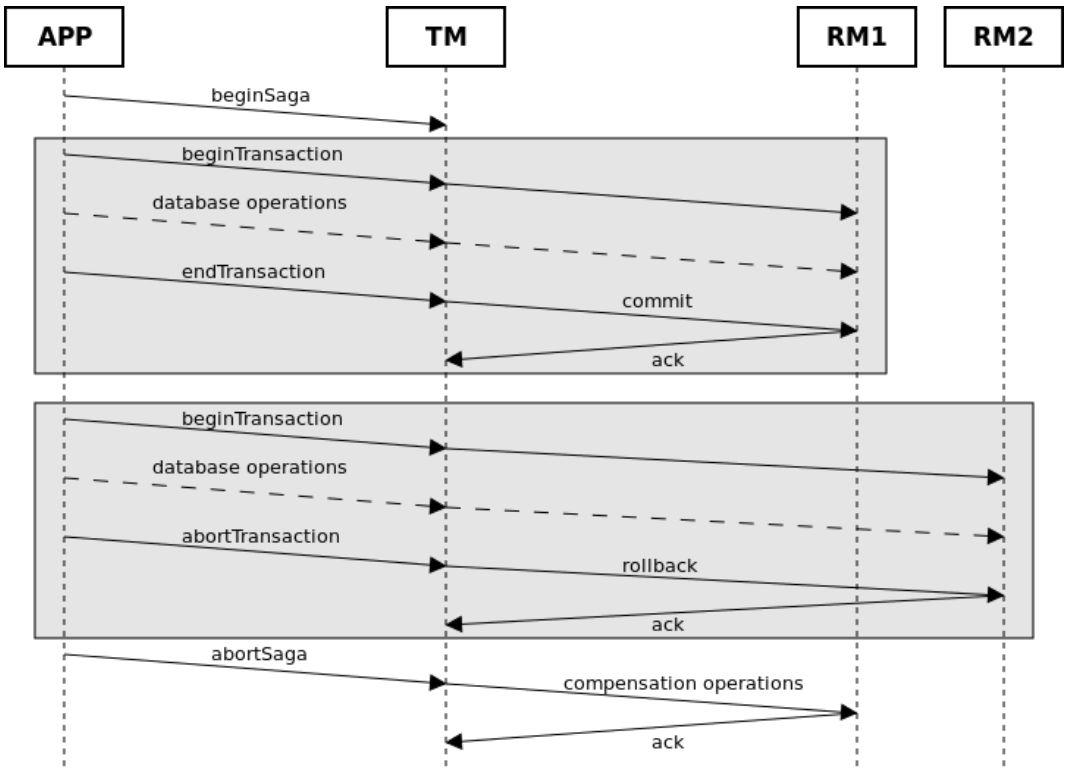

Figure 2.5: Application (APP) which executes a Saga with two sub transactions

Figure 2.5 shows the message flow of a distributed Saga consisting of two sub transactions. The first sub transaction is successful and therefore the local commit on *RM1* is executed immediately. The second sub transaction is aborted by the application and therefore a rollback is executed on *RM2*. Afterwards the application decides to abort the

whole Saga and the transaction manager executes the compensation actions on *RM1*. The diagram also shows that sub transactions are executed sequentially if the application does not use threading. In a Saga all sub transactions can be ACID compliant, but the whole Saga need not.

The Saga model has been the first transaction model which was using compensation across systems to ensure eventual consistency. Although it has been originally designed for database management systems, its pattern including compensations has been used by many other transaction models like the Flex model or web service compositions (see next sections). The advantage of this pattern is that LLTs can be executed without long-term locking of objects. This improves concurrency and performance of the system, however there is also a drawback: Sagas are not ACID compliant because the isolation property has been relaxed.

Sagas have not been directly implemented in any established application server or middleware. However the Saga pattern has been indirectly implemented with the transaction models in the next sections.

2.2.5 Flex Transactions

The Flex Transaction model [BEeK93] is based on the Saga model but extends it in several points. It can be seen as a kind of workflow model where several sub transactions are executed in a pre-defined way. A transaction is described in the InterBase Parallel Language (IPL). Like in the Saga model every sub transaction can be committed right after its completion. When the parent transaction aborts although the sub transaction has already committed, then its (optional) compensation action is executed to undo the action. One advantage to the Saga model is that several sub transactions can be executed in parallel. Dependencies can be used to specify the execution order among sub transactions. There is also a main difference in the way how transactions can be used by an application. In the Flex Transaction model the global transaction is defined in a single IPL file and is entirely passed to the transaction manager. The transaction manager then executes the sub transactions and takes care of the coordination among them. Whereas in the Saga model the coordination among sub transactions is accomplished in the application. After completion the result of the transaction is returned to the application. The definition of a Flex transaction consists of three sections:

- variables and data structures
- definition of sub transactions
- description of the dependencies among sub transactions

IPL supports several primitive data types like **int**, **real**, **boolean**, **charString** but also supports the possibility to compose more complex data structures. The definition of a sub transaction contains the following parameters and sections:

- input parameters and output parameters
- the name of the system system and the network name of the remote machine
- the body which contains all operations
- the body of commit operations, which are executed when the global transaction has decided to commit
- the body of undo operations, which are executed when the sub transaction has already been executed but the global transaction has decided to abort

It can be specified whether sub transactions are immediately committed after they have completed or if the commit is deferred until the final decision of the global transaction. The undo statements are executed if the sub transaction has been committed and the final decision is *abort*. If a sub transaction needs to be compensated, then only the undo operations of the sub transaction itself are executed, rather than all undo operations of its sub transactions. All statements within a sub transaction are executed at a single system, whereas different sub transactions can operate on different systems.

Listing 2.1 shows a Flex transaction which consists of three sub transactions. The example in the listing models function replication using sub transactions *performDebit1* and *performDebit2*. If *performDebit1* fails, then *performDebit2* is executed. If one of the two sub transactions has been successful, sub transaction *performCredit* is executed. All sub transactions require an input of type *inparams* which contains the account number as well as the amount which should be transferred. The listing also shows two different commit strategies. Sub transactions *performDebit1* and *performCredit* are using a deferred commit which is performed when the global transaction is committed. In contrast sub transaction *performDebit2* immediately commits and therefore a compensation action has to be specified which undos the already committed operations.

```
1  program
2      record inparams of /* the inputs from the application */
3          accNum : charString; /* account number */
4          amount : integer; /* the amount of money */
5      endrecord;
6
7      record outparams of /* the output of the sub transactions */
8          accNum : charString; /* account number */
9          balance : integer; /* the new balance */
10     endrecord;
11
12     /* define input "in" of type inparams */
```

```
13      input in : inparams endinput;
14
15      subtrans performDebit1 (in) : outparams use sybase at site1
            output
16          beginexec /* execution step, in SQL format */
17              begin tran txDebit1;
18              update bank set balance = balance - $$in.amount$$ where
                    accNum = $$in.accNum$$;
19              select accNum,balance from bank where accNum = $$in.
                    accNum$$;
20          endexec
21          beginconfirm /* confirm step, in SQL format */
22              commit tran txDebit1;
23          endconfirm
24          beginundo /* undo step, in SQL format */
25              rollback tran txDebit1;
26          endundo
27      endsubtrans;
28
29      subtrans performDebit2 (in) : outparams use ingres at site2
            output
30          beginexec /* execution step, in SQL format */
31              update bank set balance = balance - $$in.amount$$ where
                    accNum = $$in.accNum$$;
32              select accNum,balance from bank where accNum = $$in.
                    accNum$$;
33          endexec
34          beginundo /* compensation, in SQL format */
35              update bank set balance = balance + $$in.amount$$ where
                    accNum = $$in.accNum$$;
36          endundo
37      endsubtrans;
38
39      subtrans performCredit (in) : outparams use sybase at site3
            output
40          beginexec /* execution step, in SQL format */
41              begin tran txCredit;
42              update bank set balance = balance + $$in.amount$$ where
                    accNum = $$in.accNum$$;
43              select accNum,balance from bank where accNum = $$in.
                    accNum$$;
44          endexec
45          beginconfirm /* confirm step, in SQL format */
46              commit tran txCredit;
47          endconfirm
48          beginundo /* undo step, in SQL format */
49              rollback tran txCredit;
50          endundo
51      endsubtrans;
```

```
52
53      dependency
54          performDebit1 : performCredit;
55          not performDebit1 : performDebit2;
56          performDebit1 or performDebit2 : performCredit;
57          performCredit : accept;
58      enddep;
59  endprogram
```

Listing 2.1: Definition of a Flex transaction using IPL

Figure 2.6 shows the message flow of the distributed Flex transaction which is shown in listing 2.1. It has been assumed, that sub transaction *performDebit1* has been successful and therefore the execution of *performDebit2* is skipped. After the successful execution of sub transaction *performCredit* the commit operations of *performDebit1* and *performCredit* are executed.

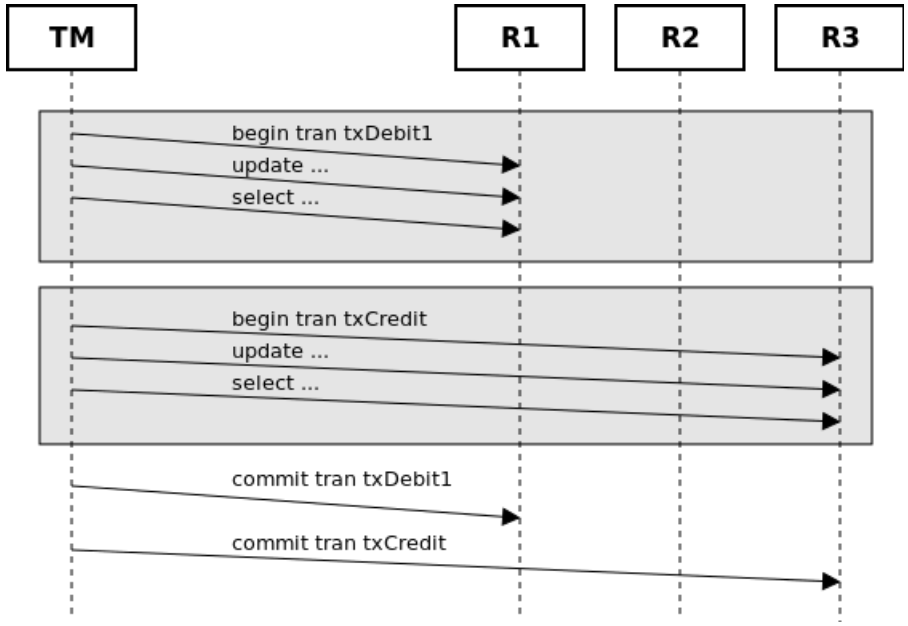

Figure 2.6: Message flow of the Flex Transaction which is defined in listing 2.1

The Flex Transaction model also supports the execution of arbitrarily nested transactions in one single atomic transaction. This can be achieved by only using sub transactions with deferred commit statements. This is a further difference to the Saga model where only sub transactions are atomic, but not the global transaction.

The Flex Transaction model has been implemented in the Corso middleware [eK05a] [eK00] [WeKT00] which consists of the Coordination Kernel [KN98] and programming

interfaces. Corso uses a virtual shared memory to coordinate distributed applications on heterogeneous environments.

2.2.6 WS-BPEL

The Web Services Business Process Execution Language (WS-BPEL, short BPEL) [7] is the current standard for expressing web service compositions. It has been specified by OASIS[5].

A BPEL definition is an XML file which defines activities to be performed and how they depend on each other. An activity can be the invocation of a web service or any other component which is required to define a business workflow. BPEL supports simple manipulation of data so that data can be adapted to the input requirements of the next activity. Further primitive structured-programming constructs like *if-then-elseif-else* or *while* are supported and activities can be executed sequentially or in parallel. Also BPEL provides the possibility to define event-handlers, fault-handlers or compensation-handlers.

A business process is separated into four major sections:

- `<partnerLinks>`: Definition of all parties (e.g. suppliers, customers, providers, ...) that interact with the process.
- `<variables>`: This section defines all variables which are used to exchange messages among the parties.
- `<faultHandlers>`: This section contains all fault-handlers which will be called in response to a fault resulting from the invocation of a service.
- `<processes>`: This section contains all activities which are required to fulfil the business process. The composition among the activities of the schedule can be defined with directives like `<sequence>` (all activities will be executed sequentially) or `<flow>` (parallel execution). Additionally the directive `<links>` can be used to map dependencies between activities within a `<flow>` section. For example a link can be used to specify that activity *A* must not be executed before activities *B* and *C* have been completed.

The creation of a new process instance is triggered implicitly by a start activity. Every executable business process must have at least one start activity which has to be annotated with the `createInstance` property. This activity can either be a `<receive>` (waits for a message to arrive) or `<pick>` (waits for one of several messages to arrive) statement.

[5] `https://www.oasis-open.org`

A business process is executed by a BPEL engine which takes an XML definition file as input. Neither code nor other input files are required. This is accomplished by the limitation that BPEL only supports web services with a proper WSDL[6] definition. If other systems want to participate in a business process then they need to be wrapped into a compliant web service. The BPEL engine is the central part of the execution and acts as the transaction manager.

Here is a list of some of the must popular engines:

- **BizTalk Server**[7] (Microsoft)
- **Oracle BPEL Process Manager**[8] (Oracle)
- **SAP Exchange Infrastructure**[9] (SAP)
- **WebSphere Process Server**[10] (IBM)

All of them are commercial products and not free of charge. However there are also Open-Source products available like **Apache ODE**[11]. The internal message flow of BPEL engines is similar to the one of the Flex Transaction model (figure 2.6).
BPEL is based on the Saga model where transactions are mapped to business processes and sub transactions are mapped as activities. Every activity operates on a single host, which can execute the activity in a single ACID compliant local transaction. Since several activities can operate on several hosts, business processes are per definition distributed transactions. The advantage of BPEL is that activities can be executed in parallel and long running transactions do not affect concurrency since object locks within activities are released when they finish. BPEL processes are dedicated to web services and therefore require that all services are providing the appropriate interface. Analogous to the Saga model BPEL processes are not ACID compliant. Atomicity, consistency and isolation properties are relaxed because the work of business processes is divided into several activities and the result of activities can be seen by other business processes as soon as they have finished.

2.3 Comparison of transaction models

Table 2.3 shows a matrix with the differences between existing transaction models with regard to several criteria. The last two models in the table are part of this thesis and will be described in detail in section 4. The following characteristics are compared:

[6] `http://www.w3.org/TR/wsdl`
[7] `http://www.microsoft.com/biztalk/en/us/default.aspx`
[8] `http://www.oracle.com/technology/products/ias/bpel/index.html`
[9] `http://www.sap.com/platform/netweaver/components/xi/index.epx`
[10] `http://www-306.ibm.com/software/integration/wps/`
[11] `http://ode.apache.org/`

- **Synchronous**. The model supports a blocking commit method.
- **Asynchronous**. The model supports a non-blocking commit method.
- **ACID compliant**. ACID compliant transactions can be performed.
- **Sub Transactions**. A transaction can consist of several sub transactions. In the BPEL model sub transactions are called activities.
- **Dependencies between sub transactions**. The execution order of sub transactions can be defined by specifying dependencies among sub transactions.
- **Parallel sub transactions**. Several sub transaction can be executed in parallel.
- **Nested transactions**. Transactions can be nested.
- **Compensation**. Compensation actions can be specified which semantically undo the effects of already committed transactions.
- **Function Replication**. If a sub transaction fails, then another sub transaction can take over and the parent transaction succeeds although a sub transaction has failed.
- **Transaction Coordinator**. The system which is responsible to coordinate and execute transactions, sub transactions and operations within transactions. This is either the application (App) or the transaction manager (TM).
- **Single Point of Failure**. If this resource fails, then the *commit* service becomes unavailable. This means that the service blocks until the resource has recovered. It is important to know that this state is different to a state where the *commit* service is available but always decides *abort*. Image that a resource manager crashes during the commit cycle of Paxos Commit. After the timeout has elapsed in the acceptor process, the transaction will be aborted. Since the *commit* service remains available, the resource manager is not considered as the single point of failure.

The already existing transaction models in the table can be separated into two groups:

- **Atomic commit protocols** - These protocols (2PC, 3PC, Paxos Commit) support ACID compliant distributed transactions. The transaction manager of these protocols is only responsible to coordinate the distributed commit. The execution of transactional operations is coordinated by the application.

	2PC, 3PC	Paxos Commit	Saga	Flex Transaction	BPEL	Ext. Paxos Commit	REXX Transaction
Synchronous	x	-	x	x	x	x[a]	x
Asynchronous	-	x	-	-	x	x	x
ACID compliant	x	x	x[b]	x[b]	x[b]	x	x
Sub Transactions	-	-	x	x	x	x	x
Dependencies between sub transactions	-	-	-	x	x	-	x
Parallel Sub transactions	-	-	-	x	x	-	x
Nested Transactions	-	-	-	x	x	-	x
Compensation	-	-	x	x	x	-	x
Function Replication	-	-	-	x	x	x	x
Transaction Coordination	App	App	App	TM	TM	App	TM
Single Point of Failure	TM	-	TM	TM	TM	-	TM[c]

Table 2.1: Comparison of transaction models

[a] returns when the result of Paxos is available, but does not guarantee that the RM already performed the commit/abort

[b] only within sub transactions

[c] in a space-based middleware like XVSM the space is the single point of failure if it is not replicated.

- **Relaxed transaction models** - These models have relaxed the ACID properties to achieve advantages in performance and functionality like nested sub transactions, parallel sub transactions or function replication. The transaction manager is responsible for the scheduling and execution of the sub transactions.

The Saga model is somewhere between these groups because although it is not ACID compliant it does neither provide features like parallel sub transactions nor function replication.

The BPEL transaction model provides very powerful features, but since it is dedicated to the integration of web services it can not be used for a space-based middleware. Also the Flex model already supports a very flexible relaxed transaction semantics and provides features like function replication or the parallel execution of sub transactions. However the Flex model has been designed for the integration of autonomous legacy systems, especially for multi database systems (MDBS) and therefore it can not be directly implemented into a space-based middleware like XVSM. A new transaction model has to be developed which is dedicated to space-based middlewares and which provides a relaxed transaction semantics like BPEL or the Flex model. Furthermore it has been decided to use Paxos Commit as part of the new transaction model to support ACID compliant distributed transactions. Paxos Commit is used because of its advantages in terms of consistency and availability compared to the other atomic commit protocols.

CHAPTER 3

The XVSM middleware and MozartSpaces

This chapter provides basic information which is required for the understanding of the next chapters. The concept of XVSM (eXtensible Virtual Shared Memory) has been developed at the Institute of Computer Languages of the Vienna University of Technology in 2005 [eK05b].

MozartSpaces[1] is the Java-based reference implementation of XVSM. The development has started in 2006 and a complete rewrite (version 2) has been released in 2010 [Bar10] [Dö11]. Since then several additional modules and functionalities have been developed which lead to a steady improvement of the MozartSpaces implementation. The current stable release is version 2.2 which already includes Security (Authentication and Authorization) [CK12] as well as persistency [Zar12].

3.1 XVSM Overview

XVSM is based on the Linda model which has been introduced by David Gelernter in 1985 [Gel85]. The formal model of XVSM as well as an implementation in Haskell is described in [Cra10]. It uses a central memory where all data is stored. Distributed application can access the memory and manipulate data with only a very limited number of operations. The most important operations are `write` (creates new data) and `take` (consumes and returns data).

Figure 3.1 shows the architecture of the XVSM middleware. The main part of the middleware is the XVSM core, which manages the access to the space and also provides the Core API. Applications can only access the space by using this interface. XVSM

[1] `http://http://www.mozartspaces.org/`

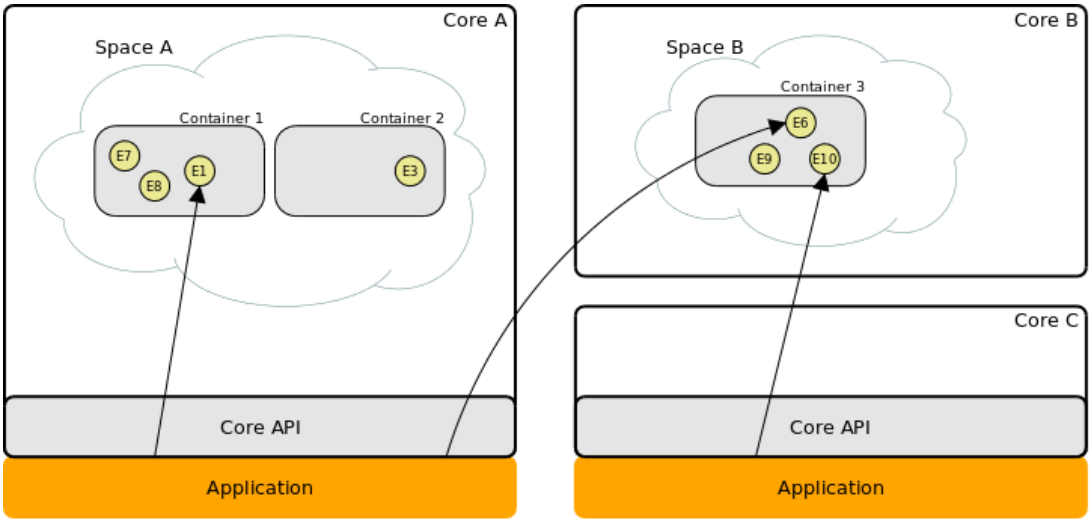

Figure 3.1: Application accessing XVSM spaces by using the provided Core API

cores can either be started without embedded space (*Core C*) or with exactly one embedded space (*Core A* and *Core B*). As shown in the figure applications can also use the local Core API to access data from a remote space.
The memory (space) of XVSM is organized in three hierarchies. The *space* is the biggest data structure. Several spaces can be hosted on the same location and every space has a separate XVSM core. Every space contains zero or more *containers* and every container stores several *entries*, which are the smallest unit of data. Entries can not be directly written into a space, but only into containers.

Every container in the XVSM framework is coordinated by one or more *coordinators*. They store additional meta information for every entry which is later used by the corresponding *selector* when an entry is selected (e.g. by operations *take* or *remove*). Supported coordinators are for example the *Key Coordinator*, which stores an unique key for every entry or the *FIFO Coordinator*, which provides a FIFO (First in, First out) ordering of the entries. Figure 3.2 shows one container which is managed by those two coordinators.

The focus during the design of XVSM has been put on an extensible and flexible architecture. The core is based on a microkernel architecture and has been designed to run on a single site. Additional functionality like distributed algorithms can be implemented on top of the core by using so-called *aspects* without much effort. An aspect contains code which can be executed before and/or after every operation. This can either be a container operation like `write` or `take` or any space operation like `createContainer`. The execution of the code can be performed within the context of the operation's transaction. Additionally aspects can also execute operations on

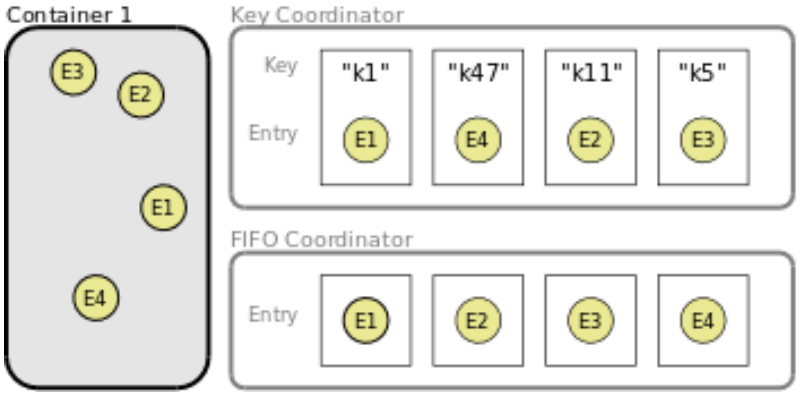

Figure 3.2: Container with a *Key Coordinator* and a *FIFO Coordinator*

remote spaces but in this case a separate transaction is needed.

3.2 MozartSpaces

This section focuses on dedicated parts of the current MozartSpaces implementation which are required for distributed transactions.

3.2.1 Transaction Support

MozartSpaces supports local transactions where all operations are performed on a single space. Transactions are compliant to the ACID properties (see section 2) and use pessimistic locking to fulfil the isolation property.

There are four isolation levels defined in the ANSI/ISO SQL standard [ISO92]:

- **Read uncommitted** - Uncommitted changes can be seen by other transactions. This isolation level allows dirty reads.
- **Read committed** - Only committed changes can be seen by other transactions. But there is no guarantee that the changes are still there when the transaction commits (*phantom read*). Dirty reads are not possible in this level.
- **Repeatable read** - Read locks are used to guarantee that data which has been read by the transaction is not modified or deleted until the commitment of the transaction has been completed.
- **Serializable** - Transactions operate in a full isolated environment and do not see other transactions. It appears that all transactions are executed sequentially. This is the strictest isolation level.

In version 2.2 of MozartSpaces the isolation levels **Read committed** as well as **Repeatable read** are supported [Dö11]. Local transactions are using timeouts to limit the execution time. When the timeout of a transaction expires a rollback is performed and all further operations which are using this transaction will fail.

Since the local transaction system of MozartSpaces is ACID compliant, it is possible to implement an ACID compliant distributed transaction system on top of the existing implementation.
MozartSpaces uses the term "sub transaction" for internal system transactions which are required to perform small operations within the core. However these sub transactions are not related to the sub transactions which are discussed in this work.

3.2.2 Persistence Support

Persistency has been introduced in MozartSpaces version 2.2 [Zar12].

The following persistency profiles are supported:

- **Lazy** - This profile uses an internal buffer in memory and writes the data into stable storage when the buffer is full. The advantage is a good performance because data is only written in big chunks. However already committed data will be lost when the system crashes before the data has been written on disk.
- **Transactional** - The database log is written to disk whenever a transaction is committed. However it is possible that the data is lost during a system crash because current storage systems are using an internal volatile buffer for performance reasons.
- **Transactional with fsync** - The *fsync*[2] command forces a storage system to flush the data of the internal buffer. This command is executed synchronously when a transaction is committed. This assures that committed data is not affected by a system crash. This is the slowest profile.

In the current implementation the following components are not covered by the persistency:

- The isolation manager including its log items and lock items.
- The internal counter of the next transaction ID.
- All space and container aspects.

[2] `http://linux.die.net/man/2/fsync`

Persistency in distributed transaction systems is a mandatory property which all cohorts as well as the transaction manager must provide. Otherwise the system will become inconsistent when any of the involved systems crashes.

If during the implementation phase it turns out that the persistency of any of the mentioned components is required, then there are two possibilities:

1. The current implementation of the persistency has to be extended by the required components.

2. If this does not make sense for any reason (e.g. to high effort) then an alternative approach using aspects could be chosen.

CHAPTER 4

Design and Model

The goal of the new distributed transaction model is to provide both: ACID compliant transactions and transactions with relaxed semantics. Therefore two models are proposed:

1. The *Extended Paxos Commit* model is responsible to perform distributed transactions with full ACID compliance. However it has drawbacks in terms of performance.

2. The *REXX* model is a highly concurrent relaxed transaction model, however it is not strictly ACID compliant.

Extended Paxos Commit does not depend on the REXX model and can therefore be used separately. However the REXX model depends on Extended Paxos Commit because it uses distributed transactions for the execution of sub transactions. This approach allows to execute ACID compliant distributed transactions within sub transactions and also supports transactions with relaxed semantics.

4.1 Terminology

The next sections describe the new transaction models in more detail. For a good understanding the following definitions for the different types of transactions are needed. These terms will be consistently used in the next sections.

Parent transactions are transactions which have at least one sub transaction.

Sub transactions have exactly one parent transaction.

Global transactions specify the whole transaction including all parent transactions, sub transactions and leaf transactions.

Top-level transactions are at the top of the transaction tree. Each global transaction has exactly one top-level transaction and the state of the global transaction is based on the state of the top-level transaction.

Leaf transactions are always at the bottom of the transaction tree and do not have any sub transactions. Leaf transactions are usually sub transactions, however if the top-level transaction is not a parent transaction, then the top-level transaction is a leaf transaction but not a sub transaction. Each global transaction must have at least one leaf transaction.

The general term **transaction** is used for parent transactions and sub transactions. Transactions can fulfil several roles at the same time. For example a transaction can act as parent transaction and as sub transaction, if it has a parent transaction and at least one sub transaction. Furthermore a sub transaction can also be a leaf transaction. Each transaction is either a parent transaction or a leaf transaction.

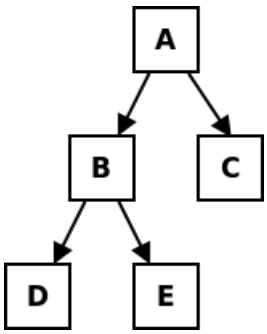

Figure 4.1: Example for a global transaction

Figure 4.1 shows an example for a global transaction. The top-level transaction is A which also acts as parent transaction. The transaction B is a parent transaction as well as a sub transaction. Transactions C, D and E act as sub transactions and leaf transactions.

4.2 Extended Paxos Commit

4.2.1 Overview

The goals of this transaction model are:

- Strict ACID compliance
- Relaxed behavior during the commit phase
- Deterministic behavior in all error cases

Table 2.3 lists three transaction models which are strictly ACID compliant: Two-Phase-Commit, Three-Phase-Commit and Paxos Commit. Due to its advantages in terms of failure scenarios Paxos Commit has been chosen to form the basis of the new transaction model. During the commit phase of Paxos Commit all cohorts have to be responsive, otherwise the transaction will fail. The goal of the following extension is to relax this strong requirement.

To accomplish this behavior Paxos Commit is extended by sub transactions[1]. Big transactions can be split into several small transactions. With that approach the original Paxos Commit is extended by two functionalities:

- **Function Replication** - Several sub transactions can be specified for the same task. During the commit phase at least one of the sub transactions must be able to commit.
- **Low Priority Transactions** - Transactions can consist of several sub transactions with different priorities. Although a sub transaction with a low priority fails, for the overall system requirements it can make sense to commit the other sub transactions which have been successful. This can be the case if the rollback of already committed sub transactions is more critical than the fact that a low priority sub transaction has been aborted.

When can the rollback and re-execution of a transaction are considered to be expensive? This depends on the individual application, however the following facts can give an indication:

- High number of operations within a single transaction
- High latency of an operation (slow connection to external systems)
- High amount of data
- Access to external systems is expensive in terms of data transfer or number of calls
- Application dependent costs (cancellation of a flight)

Figure 2.4 in the related work section shows the commit phase of the original Paxos Commit algorithm.

[1]These sub transactions are not related to the internal sub transactions of MozartSpaces (see section 3.2.1)

4.2.2 The extended algorithm

The extended algorithm allows to split a transaction into one top-level transaction and several sub transactions. For the commit of the leaf transactions the original Paxos Commit is used. This extension focuses on the relation between parent transactions and its sub transactions. Therefore the following design decisions have been specified:

- **Parent transactions** consist of one or more sub transactions. Parent transactions can not be used to execute transactional operations.
- **Sub transactions** have exactly one parent transaction. Sub transactions are either parent transactions or leaf transactions.
- **Leaf transactions** are executing their transactional operations in their own isolation context. Therefore the modifications of a leaf transaction can never be seen by other leaf transactions.
- **Commit Rules** can be used to specify how the result of a sub transaction affects the result of the parent transaction and all siblings.
- **Timeouts** are only supported for leaf transactions but not explicitly for parent transactions. The parent transaction is processed as soon as all sub transactions are either prepared or aborted.
- **ACID compliance** is preserved by committing all transactions of a global transaction in a single atomic step.

These design decisions cause modifications in the following three areas of the original algorithm:

1. **Transaction Creation** - The transaction identifier of a parent transaction must contain the identifiers of all sub transactions. When a new sub transaction is created, then the new transaction identifier must be registered in the parent transaction.
2. **Transaction Commit** - The application has to pass a commit rule which is processed by the leader process.
3. **Leader process** - The leader process waits until the *2bPrepared* messages of all sub transactions have been received. Afterwards the commit rule is applied onto the transaction results of the sub transactions. The result of the commit rule is the transaction result (*committed* or *aborted*) for the parent transaction and for all sub transactions.

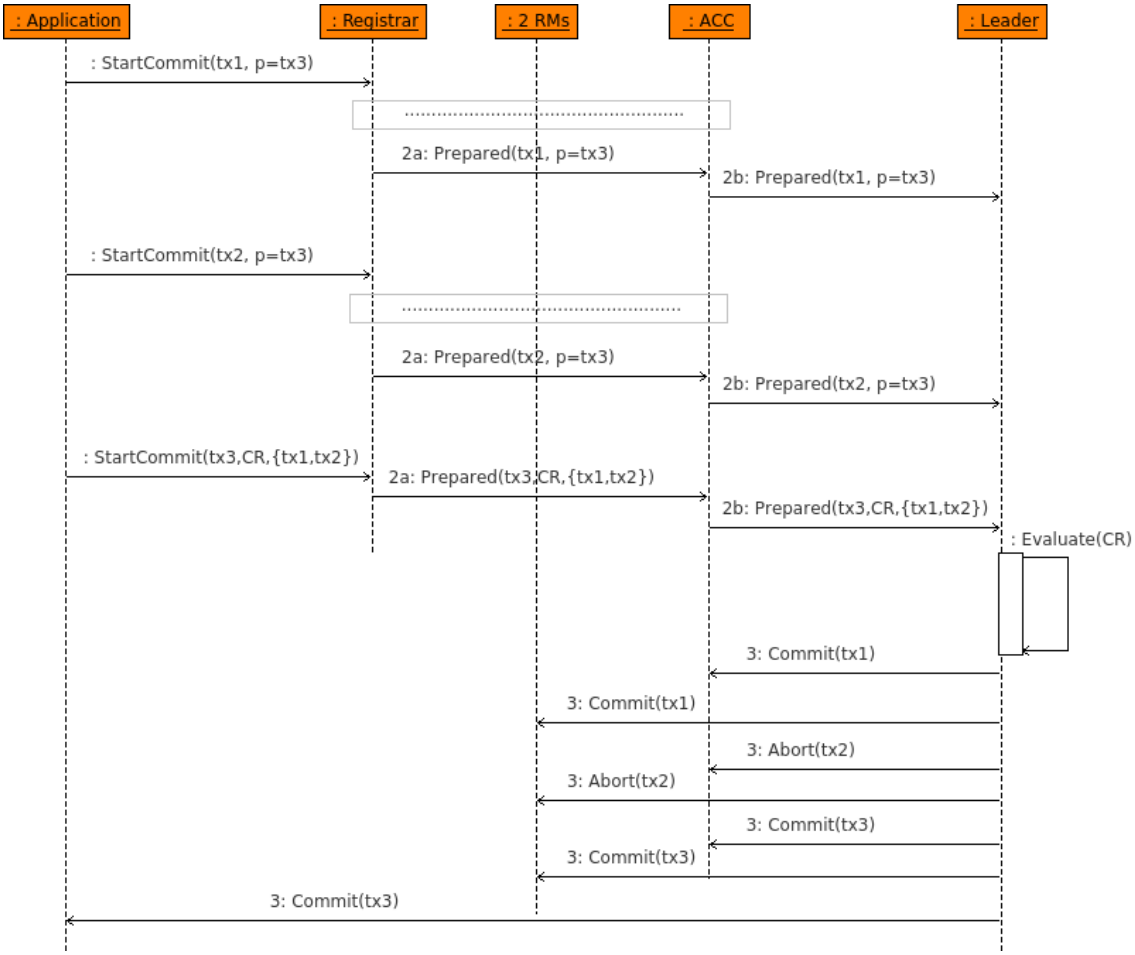

Figure 4.2: The extended algorithm applied to a parent transaction with two sub transactions

Figure 4.2 shows the sequence diagram of the extended algorithm. It shows transaction *tx3* which consists of two sub transactions (*tx1* and *tx2*). It is assumed that all resource managers as well as the registrar send *prepared* in their *Phase2a*-message. For the sake of simplicity only one acceptor is used and the messages between the registrar, the resource managers and the acceptors have been omitted. They are same as in the original Paxos Commit algorithm in figure 2.4. The extended algorithm starts when the leader process has received all *Phase2b*-messages.

When the commit phase of a sub transaction is started, the transaction identifier of the parent transaction is passed to the Paxos Commit algorithm. When the *2bPrepared* message of a sub transaction is received by the leader it first checks whether the *2bPrepared* message of the parent transaction (*tx3*) has already been received. If this is not the case, the leader waits until it has been received.

Similar to the sub transactions, the application also passes additional data of the parent transaction to the Paxos Commit algorithm: the *Commit Rule (CR)* as well as the list of all sub transaction identifiers.

Once the *2bPrepared* message of the parent transaction *tx3* has been received, the leader checks whether the *2bPrepared* messages of all sub transactions have been received so far. If yes, then the *Commit Rule* is applied onto the result of the sub transactions (*prepared* or *aborted*). The result of the function is the final status of the parent transaction as well as the final status of all sub transactions. The result of all transactions is sent to all acceptors, to all participating resource managers as well as to the application. The commit messages sent to the acceptors are not required for the functionality, however they improve the performance of the protocol if nodes have to query the result of transactions. This happens either when a node restarts or when a message has been lost and the internal timeout has been elapsed. The commit message to the application contains the result of the transaction and all of its sub transactions. The message is also used for the synchronization within the application. The results of sub transactions *tx1* and *tx2* are part of the result of *tx3* and therefore not explicitly sent to the application.

4.2.3 Timeout handling

The timeout handling for leaf transactions has not been modified in the extended algorithm. For every leaf transaction a relative timeout can be specified. The timeout is started as soon as the commit phase of the leaf transaction has been started. If the timeout elapses, the acceptor nodes will propose to abort the leaf transaction.
For parent transactions timeouts are not directly supported. Parent transactions commit as soon as the results of all sub transactions are available. So the timeout of the parent transaction depends on the timeouts of its sub transactions.

4.2.4 The global Commit Rule

The commit rule is passed by the application at the *commit()*-call of parent transactions. It determines the result of the parent transaction as well as the results of all sub transactions. Therefore it uses the intermediate results of the sub transactions as input values. The execution of the commit rule is using the following signature:

- **Input parameter:** Pair $\{tx \rightarrow IntermediateState\}$ for every sub transaction identified by tx where $IntermediateState$ is either *prepared* or *aborted*.
- **Return value:** Pair $\{tx \rightarrow FinalState\}$ for every transaction (parent and sub transactions). tx is the transaction identifier of the transaction and $FinalState$ is either *committed* or *aborted*.

The commit rule can be seen as the relation

$$S \times I \mapsto S \cap \{p\} \times F$$

where S is the set of sub transactions, p is the parent transaction, $I = \{prepared, aborted\}$ and $F = \{committed, aborted\}$. Three different types of commit rules are supported:

- **Logical Commit Rule** - A boolean expression is used to determine the result based on the results of the sub transactions.
- **Threshold-based Commit Rule** - The transaction is successful if the number of successful sub transactions is above a certain threshold.
- **Default Commit Rule** - The transaction is only successful if all sub transactions are successful.

4.2.4.1 Logical Commit Rule

This rule uses a boolean function to obtain the results of the transactions. Therefore it is required to map the $IntermediateState$ to boolean values:

- *prepared* $\rightarrow TRUE$
- *aborted* $\rightarrow FALSE$

The logical commit rule uses an additional optional argument. This is a ordered list of sub transactions which specifies the priorities among them. The first entry has the highest priority. The evaluation of the logical commit rule is explained with the help of an example. Therefore we make the following assumptions:

- For the example we use the transaction shown in figure 4.2.
- Result of Paxos Commit of the sub transactions: *tx1: prepared*, *tx2: prepared*
- *Commit Rule* (XOR relation)

$$res = \neg((\neg tx1 \wedge \neg tx2) \vee (tx1 \wedge tx2))$$

- Transaction *tx1* has higher priority than *tx2*

The leader performs the following actions:

1. Apply the *Commit Rule* to the result of the sub transactions:

$$\neg((\neg TRUE \wedge \neg TRUE) \vee (TRUE \wedge TRUE)) = FALSE \qquad (4.1)$$

2. When the result of Paxos Commit is *prepared*, then the leader can either decide to *committed* or *aborted*. However when the result is *aborted*, the leader has no option. The goal of the leader is to find a solution where the result is *committed* (boolean $TRUE$) and where the number of sub transactions, which have to be *aborted*, is minimal. For this purpose the leader uses a SAT-Solver [vHvHLP07] which outputs a model that satisfies the equation. For an optimal result the solver has to prioritizes $TRUE$-literals against $FALSE$-literals. Furthermore it has to consider the priorities of the sub transactions so that it first assigns $FALSE$ to the sub transaction with the least priority. In this example the leader would decide to abort sub transaction *tx2* which would satisfy the equation:

$$\neg((\neg TRUE \wedge \neg FALSE) \vee (TRUE \wedge FALSE)) = TRUE \tag{4.2}$$

3. The leader has found a solution and passes the following result to the extended Paxos Commit algorithm:

   - *tx1*: *committed*
   - *tx2*: *aborted*
   - *tx3*: *committed*

A possible use case for this commit rule would be a highly available system which is connected to two services with the same functionality. The logical commit rule could be used to wrap both services into a single redundant service which could be part of a bigger transaction. In general this functionality is called function replication.

4.2.4.2 Threshold-based Commit Rule

This rule only takes one argument - the threshold value. The result of the parent transaction is determined by the number of successful sub transactions. If this number is higher or equal to the threshold, the result is *committed*, otherwise *aborted*. The state of all *prepared* sub transactions is set to the same value as the parent transaction. A possible use case of this commit rule would be a replicated write to a huge number of remote systems where it has to be guaranteed that the write succeeds on a certain number of systems.

4.2.4.3 Default Commit Rule

This rule does not require any additional argument. The result for the parent transaction is *committed* if all sub transactions have been successful, otherwise it is *aborted*. The state of all *prepared* sub transactions is set to the same value as the parent transaction. The default commit rule is a special case of the other commit rules. It is the default rule if no explicit commit rule has been specified.

4.2.5 Failure scenarios

Due to the extension of the original Paxos Commit algorithm, there are also new failure scenarios which have to be considered:

- **Sub transactions do not finish** - Since parent transactions do not have separate timeout values, they are waiting until all sub transactions have finished. If sub transactions do not finish before their timeout, then the acceptors are responsible to abort them. This error case is already covered by the original algorithm.
- **Result of parent transaction is not available** - If the application does not commit the parent transaction or if the *startCommit*-message is lost, then the leader process is not aware of the parent transaction. The same problem occurs if the *Phase2a*- or *Phase2b*-message of the parent transaction is lost. In all of these cases the leader queries the result from the acceptors. If the acceptors are aware of the parent transactions, they return its result, otherwise they initiate an abort of the parent transaction.

4.2.6 Proof of consistency

The parent transaction as well as all sub transactions are using separate instances of Paxos Commit. The Paxos Commit instance of the parent transaction only contains the commit rule which is passed via the registrar to the acceptor processes and then to the leader process. Although no resource managers are involved in the instance of the parent, the behavior of Paxos Commit remains the same. Therefore the only modification of interest of Extended Paxos Commit related to Paxos Commit is the operation of the leader process. Since there is only one leader process we must not care about concurrency.

The consistency of Extended Paxos Commit is already guaranteed by the acceptor processes of Paxos Commit. Once all *Phase2b*-messages have been received by the leader process Paxos Commit assures that their values (*prepared*, *aborted* or the set of sub transactions) does not change. Therefore the result of the commit rule is always the same even if a new leader is elected.

4.3 The REXX Transaction Model

The term ‘REXX transaction’ stands for ‘Relaxed transaction model for XVSM’ which already indicates the relation to the XVSM middleware.

4.3.1 Overview

For the new relaxed transaction model the following design goals have been specified:

- Support of distributed transactions
- Support of long lived transactions (LLT)
- Support of asynchronous transactions
- Monitoring of the transaction state
- No support of legacy systems
- High concurrency and performance
- Focus on transactions for coordination
- Dynamic join/leave of nodes

The overall goal is to provide dependable integration patterns for the XVSM middleware.

The idea of the new relaxed transaction model is to separate one big (global) transaction into several parent transactions and sub transactions. Transactions are executed with the standard ACID properties whereas the global transaction uses a relaxed model in terms of isolation and consistency. Isolation is only guaranteed within a transaction transaction but not in the context of the global transaction. Therefore it is possible that other transactions see and modify an intermediate result of a transaction. The consistency property is temporarily violated because other transactions can see inconsistent states. However, eventually the global transaction will be consistent.

Transactions commit immediately when they are finished. Nevertheless it can be necessary to undo the transaction's operations if one of the other transactions fails. For that case every transaction can be equipped with a compensation action. The compensation action is used to semantically perform an undo of the already committed operations. The REXX model is aligned to the Flex model (section 2.2.5), which already supports many of the discussed features in section 2.3. However the REXX model provides two additional features:

- Support of asynchronous commit and
- distributed ACID compliant transactions.

Possible use cases are high-performance business workflows with various activities which have to be processed with a certain degree of consistency. Imagine the following activities which have to be performed by a travel agency when it is booking a trip:

1. Book a flight
2. Book a taxi from the airport to the hotel
3. Book a hotel
4. Book an event (either a sightseeing tour or a ticket for a musical)

With REXX transactions the booking of the flight, the taxi and the hotel can be executed in parallel. For the booking of the event function replication is used so that if the sightseeing tour is already fully booked, a ticket for a musical is purchased.

Each activity only locks the objects which are required for its task and the locks are released as soon as the activity is completed. Therefore if the booking of the flight takes longer, the object locks of the other activities might have already been released. This allows a higher concurrency than an atomic commit protocol like Paxos Commit.

4.3.2 Assumptions and Definitions

The idea of the REXX model is to specify a global transaction including parent transactions, sub transactions and their internal communication in advance. Afterwards the global transaction is handed over to the transaction manager which will then take care of the coordination. All transactions (parent and sub transactions) are executed individually by the transaction managers. After the global transaction has been completed the result will be passed back to the application. This keeps the developer API simple. For the REXX transaction model the following design decisions have been defined:

- **Space-based design:** The model should be designed to run on a spaced-based middleware. Therefore the focus is set on a data-driven communication.
- **Legacy Systems:** The model does not support legacy systems. Therefore transactions must not directly interface legacy systems. Transaction are only allowed to operate on the corresponding space.
- **Execution environment:** The code of transactions and compensations is executed in the scope of the transaction manager. The application passes the global transaction in a fire-and-forget style to the transaction manager. The application can not influence the execution once it has started the commit of the global transaction in the user space. The concrete execution can also be performed on a space different to the space of the transaction manager. This space is called execution space and can be specified in the configuration.
- **Retry on error:** When the execution of a transaction or compensation fails for some reason, then the execution will not be repeated. The reason for this decision

is that it is unlikely that a transaction succeeds after it has failed some seconds ago.

- **Timeouts:** Developers can specify transaction timeouts for the execution of leaf transactions and for the execution of compensation actions. If the execution does not finish within that period, the result of the execution is considered as *aborted*. The correct handling of timeouts can only be guaranteed if developers are following certain rules which are discussed later. It is also supported to specify a timeout for the global transaction.

- **Nested transactions:** Transactions can be nested without any limitation of the depth. A parent transaction can consist of several sub transactions where a sub transaction itself can again act as a parent transaction. So a transaction can be both - sub transaction and parent transaction. Parent transactions can have a compensation action but do not have an execution body (transaction action). This restriction helps to keep the transaction model simple without cutting the expressiveness of transaction compositions. The state of parent transactions is based on the state of its sub transactions as well as the specified aggregation function. The aggregation function can be specified by the application. It takes the results of the sub transactions as input parameter and returns either *committed* or *aborted*.

- **Compensation Scope:** Compensation is only performed in the highest already committed transaction. Imagine a (parent) transaction which consists of two sub transactions. If the parent transaction commits it takes over the compensation responsibility of all of its sub transactions. As of this point the sub transactions are not queried for compensation any more. Of course, the compensation of the parent can call the compensation of the sub transactions, but this is not part of the model and has to be assured by the application developer. The compensation responsibility of a parent transaction does not stop at its sub transactions but also applies to sub transactions of sub transactions. So the compensation scope increases whenever a parent transaction commits. This approach for the compensation keeps the compensation effort for the coordinator on a manageable level even for transactions with several transaction hierarchies. This is the same approach which has been used in the Flex model. Imagine a travel agency which has booked a trip consisting of a flight, a hotel and a taxi. If the customer cancels the trip, the travel agency does not cancel the individual reservations of the flight, the hotel and the taxi. Instead it can sell the whole trip to another customer.

- **Dependencies:** There are two types of dependencies:

  - **Dependency between the parent and its sub transactions:** There is an implicit termination dependency between a parent transaction and all of its

sub transactions. So the parent transaction can not complete before all of its sub transactions have finished.

  - **Dependencies among transactions:** Transactions can depend on one or more other transactions as long as there is no recursion loop. In the default setting all predecessors must have successfully committed in order that a transaction can start its execution. If at least one transaction has not been successful, the transaction immediately aborts and will not be executed. However the transaction model provides an option which overrules this behavior and allows the execution of a transaction even though its predecessors have not been successful. This options allows to implement function replication within transactions. If one transaction fails, then another transaction can take over and the global transaction can succeed.

- **Compensation dependencies:** During the compensation phase transactions follow their compensation dependencies. Transactions can only depend on sibling transactions. This is not a limitation because nested compensations are not supported and therefore dependencies on other transactions would make no sense. If no explicit compensation dependencies have been specified, the execution order of the compensations is the reversed order as the execution of the transactions. A dependency is fulfilled if the predecessor has terminated, regardless of the result.

- **Parameter passing:** A global context is provided to allow communication among sub transactions and between sub transactions and the parent transaction. Sub transactions can write to the context and as soon as they commit, the written values are visible to the other sub transactions as well as to the parent transaction. The context is provided in the normal execution as well as in the compensation phase.

- **Transaction results:** Transactions can write data to the global context. The context is then passed back to the application as part of the transaction result.

- **Preemptive abort of sub transactions:** If the execution time of sub transactions exceeds the specified timeout, then the execution is aborted and a rollback is performed.

- **Crash of transaction manager:** The internal state of the transaction managers must be consistent even after a system crash. Section 4.3.6 is dedicated to that topic.

- **Crash of the execution space:** A crash of the execution space is detected during the execution of a sub transaction by the corresponding transaction manager. It then aborts the executing sub transaction and performs a rollback.

- **Crash of a resource node:** If during the execution of a sub transaction a resource node crashes, then the developer has to take care of it. If an error is not handled within the transaction's code, then the transaction manager will abort the execution and perform a rollback.
- **Crash of the application:** Applications commit transactions in a fire-and-forget style and therefore the reference to the transactions is lost when an application crashes. The transaction manager executes the transaction without recognizing the application crash. After the application has recovered it has to process the results of the transactions which have been finished during the crash. Therefore transactions can be labelled with an application identifier and can later be queried with that identifier.
- **Error in compensation:** If there is an error in the compensation action, the consistency of the global transaction is violated and the application must take care of the recovery. Sub transactions with a failed compensation will be set to state *compensation failed*. This state will then be propagated to the global transaction.
- **Availability:** The new transaction model assumes that the space on which the transaction managers operate is always available. If there is a higher degree of availability required then the space has to be replicated.

4.3.3 Architecture Overview

Figure 4.3 shows the architecture of the REXX transaction model. The centralized space holds several data containers which are used for the coordination and communication between all components. As already mentioned the term transaction is used for parent transactions and sub transactions. Global transactions consist of several transactions, compensation actions and the dependencies among transactions. They are completely defined within the application and are then passed to the transaction managers which will take over responsibility. The transaction managers execute the individual transactions at the execution space and when the global transaction is finished the result can be retrieved by the application. Through the normal API applications can only see the result of global transactions if the global transactions are already finished, whereas the Monitoring API (MAPI) provides a detailed view of the internal data of the transaction manager. It allows to query the current status of all transactions of certain global transactions even if the global transaction has not finished yet. The REXX transaction model focuses on concurrency and performance by using the provided features of a space-based middleware. Therefore the model already considers scalability for the later implementation. All transaction managers act autonomously and so it is possible to run several instances in parallel without additional coordination overhead. The number of instances can be adapted to the expected load.

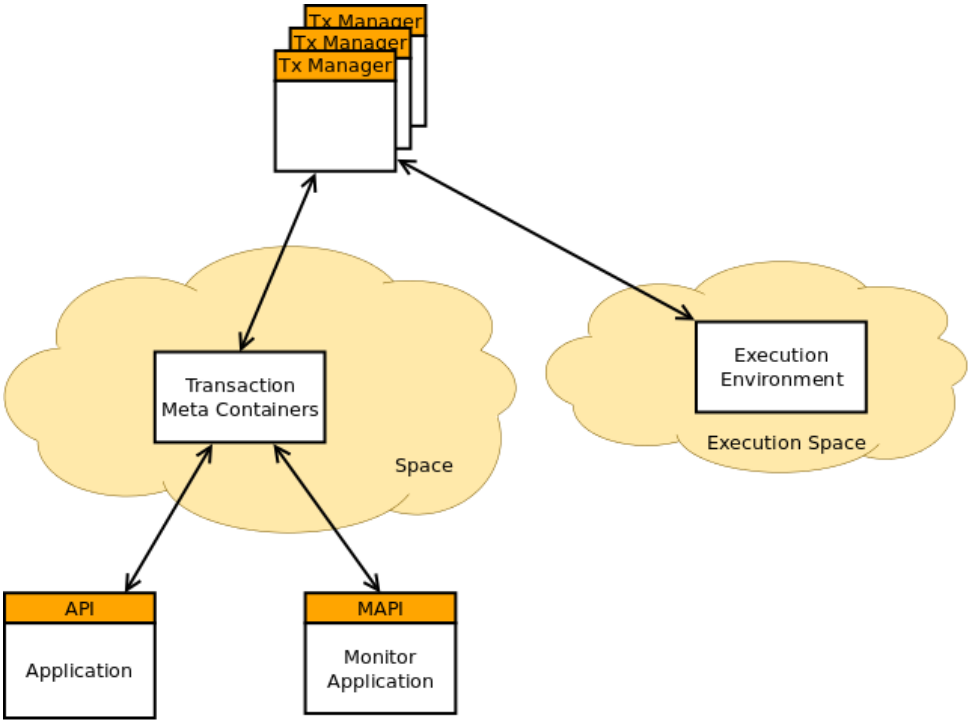

Figure 4.3: Architecture overview

4.3.4 Provided APIs

4.3.4.1 Standard API

The REXX transaction model provides the following methods in its standard API:

- `commitTransactionAsync` asynchronously hands over a global transaction to the transaction manager and returns the transaction reference immediately. This reference can later be used to query the result. Furthermore a callback function can be specified which will be executed as soon as the global transaction has completed. The return value is the result the global transaction as well as the results of all of its transactions.
- `waitForTransaction` blocks until a certain global transaction has been completed and returns the result. A timeout can be specified to limit the maximum waiting time.
- `commitTransaction` is a combination of `commitTransactionAsync` and `waitForTransaction`.

- `getOpenTransactions` returns the transaction reference of all running and completed global transactions whose result has not been queried so far. The method `waitForTransaction` can then be used to retrieve the actual results.

The global transaction in the application space contains the parent transaction, all sub transactions as well as the related compensation actions and other options like timeouts and dependencies. The main task of the `commitTransactionAsync` method of the API is to preprocess the transaction which has been passed by the application so that the transaction managers can handle it appropriately. Therefore it splits the transaction into its sub transactions. The transaction manager then only operates onto the sub transactions. Additionally the predecessors of parent transactions have to be propagated to all leaf transactions. This is because the internal model only supports dependencies for leaf transactions. Parent transactions must not have any dependencies. Imagine parent transaction A which consists of two sub transactions B and C. If A depends on another sub transaction D, then new dependencies have to be added so that B depends on D and C depends on D. After the execution of the global transaction the method `waitForTransaction` collects the results of all corresponding sub transactions and returns them to the application.

4.3.4.2 Monitoring API (MAPI)

The monitoring API can be used by external monitor applications to obtain the internal status of the transaction manager. This status consists of the states of the sub transactions as well as their dependencies, context variables and error messages. The MAPI provides the following methods:

- `getAllTransactions` returns all global transactions of the transaction manager. This does not include global transactions whose result has already been queried because all internal data related to a transaction is deleted as soon as the result has been queried.
- `getRunningTransactions` returns only global transactions which have not been finished yet.
- `setNotificationListener` can be used to specify a callback method which will be called whenever the internal state of the transaction manager changes.

4.3.5 The Transaction Manager (TM)

The transaction manager is the core component of the REXX transaction model. It looks for waiting transactions and only operates on one transaction at a time. The transaction manager does not know the global transaction which has been handed over by

the application. It only works on data which is needed for the processing of the current transaction. This data contains

- the parent transaction (if exists),
- all sub transactions (if exist),
- the code for the execution and compensation,
- all predecessor transactions,
- all successor transactions and
- the available context variables.

The state of the transactions is internally stored in the following containers:

- `data` - Every transaction is stored as a single entry in this container. The API is responsible to split global transactions from the application into its sub transactions and parent transactions and to write them into the data container. Transactions also contain the reference to the execution and compensation actions. For a global transaction with two sub transactions the container would contain three entries.
- `context` - This container is used for the intercommunication between transactions as well as for passing back data to the application.
- `dependencies` - This container stores all dependencies between transactions - normal dependencies as well as compensation dependencies.
- `meta` - This configuration container stores runtime data like the next free transaction identifier.

All of these containers are placed in a persistent storage so that their data is not lost when the system crashes. The data containers are used to coordinate the access between the transaction managers and the applications. All containers can be concurrently accessed by several transaction managers and applications.

4.3.5.1 Transaction States

This section describes the states and transitions between transactions. The section focuses on transactions and not on the global transaction. The state of the global transaction is equal to its top-level transaction. The transaction states can be divided into two types:

- **final states** - For the application only those states are visible through the API. There are four different final states:
  - *committed* - The transaction has finished successfully.
  - *aborted* - The transaction has not finished successfully.
  - *compensated* - The transaction has been successfully compensated after it has been committed.
  - *compensation failed* - The compensation of the transaction has failed. This state means a severe error since it can cause an inconsistent system. If a transaction has to be compensated but no compensation action has been specified then it is assumed that no compensation is required and the state is set to *compensated*.
- **intermediate states** - These states are required for the internal processing of the transaction and are therefore never visible to the application. However transactions in intermediate states can be queried with the monitoring API (section 4.3.4.2). Here is the list of all intermediate states:
  - *waiting for execution* - The transaction is waiting for a free transaction manager.
  - *suspended* - The transaction depends on at least one transaction which has not been completed yet.
  - *running* - The transaction is being executed. This state is required since long-lived transactions are supported and therefore the execution of transactions can take several hours or even days. Otherwise is would not be possible to distinguish between leaf transactions which are waiting for a transaction manager and leaf transactions which are currently executed by a transaction manager.
  - *waiting for compensation* - The transaction is waiting for a free transaction manager which processes the compensation.
  - *compensation suspended* - The compensation depends on at least one compensation which has not been completed yet.
  - *running compensation* - The compensation is being executed. This state is required since long-lived transactions are supported and therefore the execution of the compensation can take several hours or even days.

Regarding the internal handling there are the following differences between parent transactions and leaf transactions:

- Although parent transactions have compensation actions like leaf transactions, they do not execute normal transaction operations.
- Due to this restriction parent transaction can never be in state *running*.
- The state of the parent transaction depends on its sub transactions and the aggregation function, whereas the states of leaf transactions depend on their predecessors and on their individual execution results.

Figure 4.4 describes the state diagram of a single leaf transaction. The initial state of all leaf transactions is *waiting for execution*. If not all predecessors are in a final state, the leaf transaction is set to *suspended*, otherwise it is set to state *aborted* or *running* depending on the evaluation of the predecessors. If leaf transactions are in state *suspended*, they are set back to *waiting for execution* if any of the predecessors has completed. From state *running* the leaf transaction either changes into state *committed* (if it has been successful) or *aborted* (otherwise). If a committed leaf transaction is scheduled for compensation it is first set to *waiting for compensation* so that the next free transaction manager will process it. Before the compensation is started the compensation dependencies are checked and if not all compensation predecessors are in a final state the leaf transaction is set to *compensation suspended* as long as it is reset by any of the compensation predecessors. During the execution of the compensation the state is set to *running compensation*. This is required to distinguish between leaf transactions which are waiting for a transaction manager and leaf transactions whose compensation is already executed by a transaction manager. After the compensation the state is either set to *compensated* (if the compensation has been successful) or *compensation failed* (otherwise).

The state diagram of parent transactions (figure 4.5) slightly differs because of the missing *running* state. One transaction is represented by one entry in the `data` container. The entries are created by the commit methods of the API.

4.3.5.2 Internal Architecture

The focus in the design of the transaction manager has been put on performance, scalability and simplicity. Several instances of the transaction manager can run concurrently. The more instances exist the more transactions can be processed in parallel. All transaction managers are operating autonomously and do not communicate with each other. The transaction manager is running in an endless loop and performs the activities shown in figures 4.6 and 4.7.

First we describe the activities in figure 4.6. The transaction manager waits for a transaction which is in state *waiting for execution*. If the transaction is a parent transaction then the state of all sub transactions is checked. If not all sub transactions are in a final state yet, then the current transaction will be set to *suspended*. Otherwise the

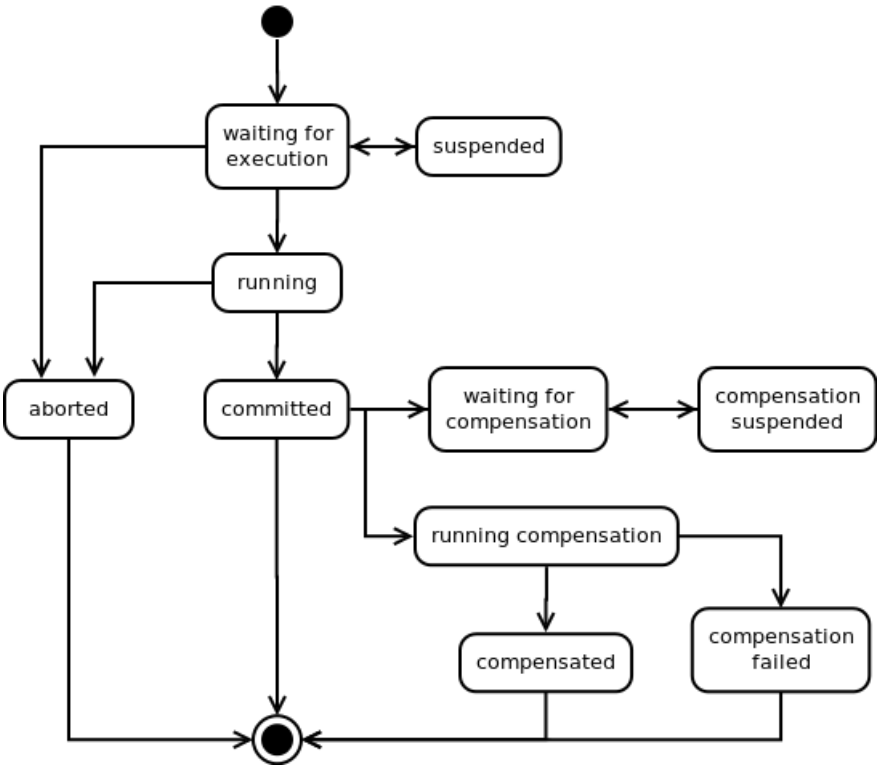

Figure 4.4: Transaction States of leaf transactions

aggregation function of the parent will be evaluated. The evaluation is performed using the following rules:

- S is the set of results of all sub transactions
- $failed(tx)$ returns `True` if sub transaction tx is in state *compensation failed*
- $aggregationFunction(S)$ is a boolean function and represents the aggregation function. The function is applied on the set of results of all sub transactions. For a transaction with three sub transactions an example aggregation function could be $(tx1 \vee tx2) \wedge tx3$ where $tx1$, $tx2$, $tx3$ identify the sub transactions. The result of a sub transaction is either *TRUE* (if the state is *committed*) or *FALSE* (otherwise). The default function is an $\wedge$-connection of all sub transactions: $tx1 \wedge tx2 \wedge tx3 \wedge .....$
- $R = \neg(\exists tx \in S : failed(tx)) \wedge aggregationFunction(S)$

If result of R is `True`, the state is set to *committed* and all dependent transactions as well as the parent transaction (if existing) will be woken up by setting their state to

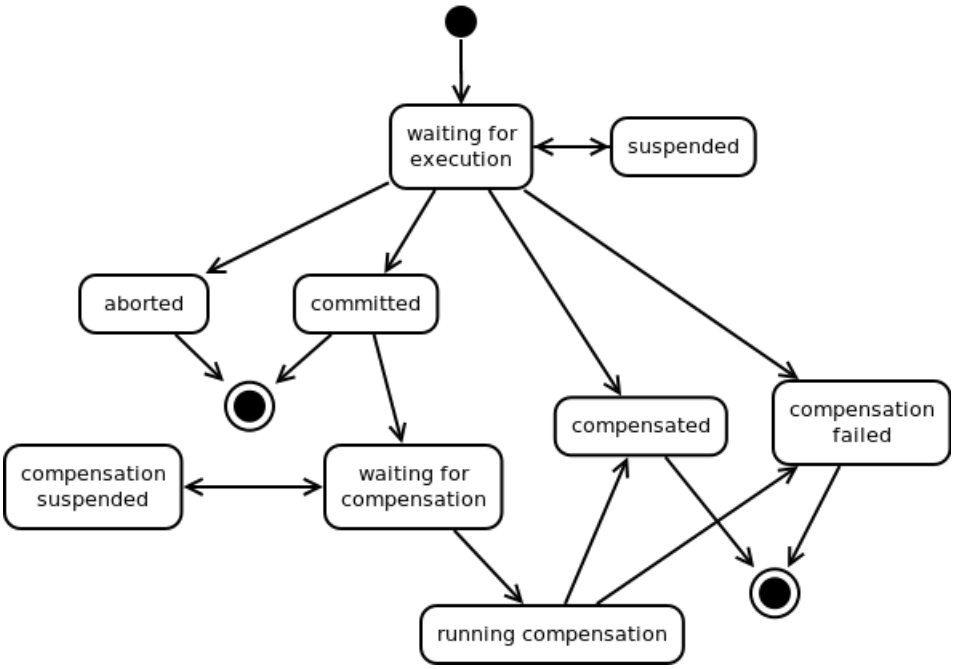

Figure 4.5: Transaction States of parent transactions

waiting for execution. If no dependent transactions and no parent transaction exist, the global transaction has been completed. If the evaluation result is `False` and there exist *committed* sub transactions, their compensations will be triggered by setting their state to *waiting for compensation* (see figure 4.7). If there is at least one sub transaction in state *compensation failed* then the parent transaction will also be set to *compensation failed*. Otherwise the state will be set to *aborted* (if all sub transactions are in state *aborted*) or *compensated* (otherwise).

With respect to these rules it can be derived that when a transaction is in state *compensation failed* then the state of the global transaction will be *compensation failed*.

Now we describe the workflow for leaf transactions. If there exist predecessors which are not in a final state the leaf transaction will be set to *suspended*, otherwise the dependencies of the predecessors are evaluated using the following rules:

- t is the current transaction
- $pred(tx)$ returns the set of all predecessors of transaction tx. If tx does not have any predecessors it returns the empty set $\{\}$
- $state(tx)$ returns `True` if tx is in state *committed*, `False` otherwise
- $failed(tx)$ returns `True` if tx is in state *compensation failed*, `False` otherwise

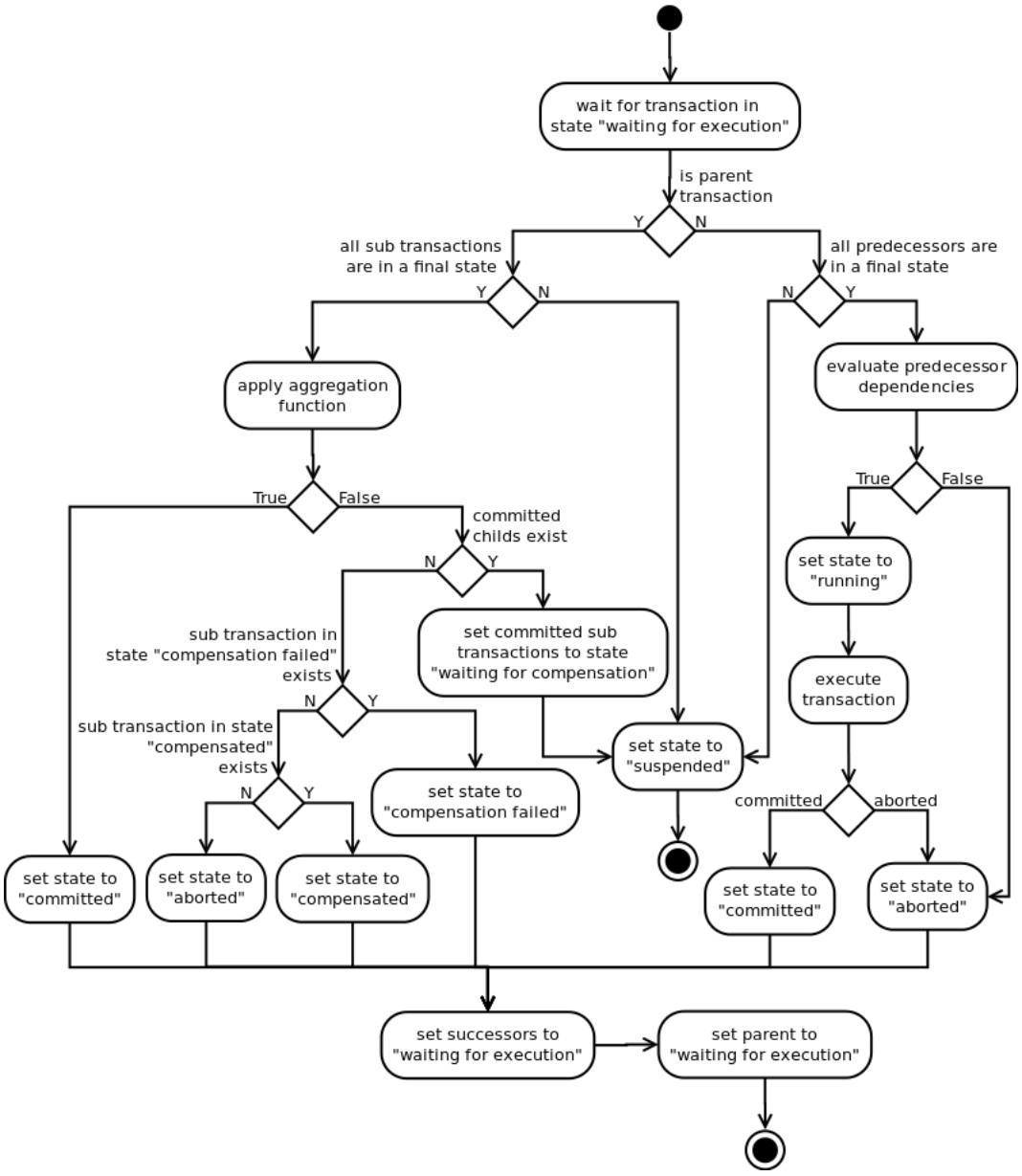

Figure 4.6: Activity diagram of the transaction manager

- $forceExecution(tx)$ returns `True` if tx is a leaf transaction and the *forceExecution* flag of tx is set. This flag can be set by the application for every leaf transaction. It forces the execution of the transaction although not all predecessors have been completed successfully.
- $R = \neg(\exists tx \in pred(t) : failed(tx)) \wedge (\forall tx \in pred(t) : state(tx) \vee forceExecution(t))$

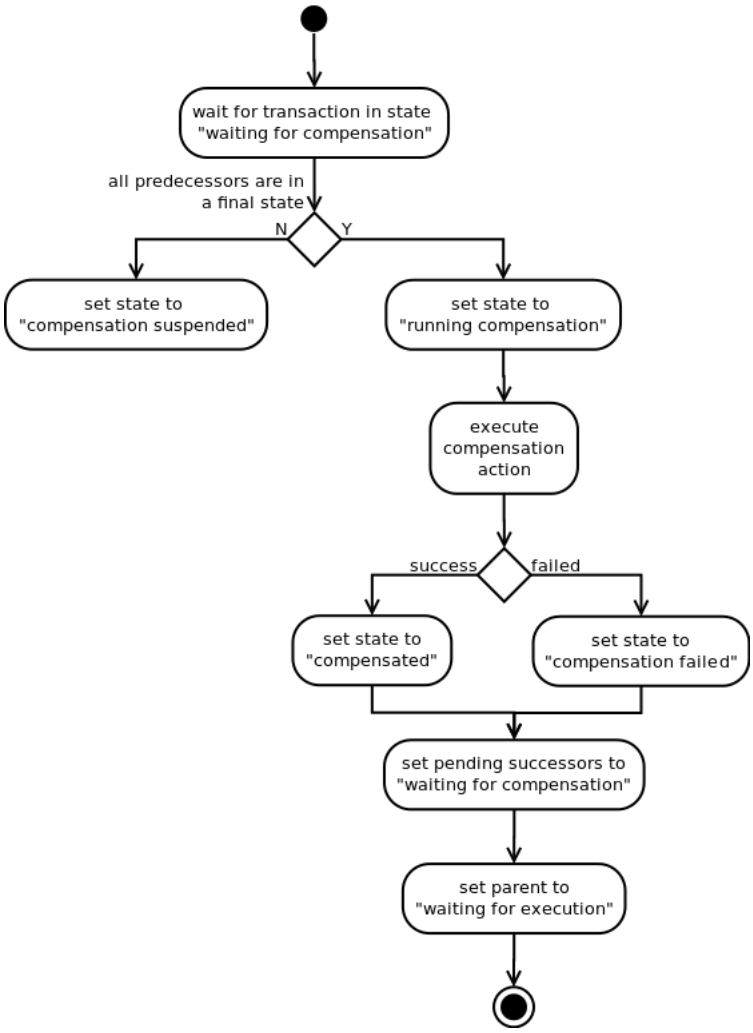

Figure 4.7: Activity diagram for the compensation handling of the transaction manager

If the result of R is `True`, then the transaction manager sets the state to *running* and executes the transaction. Otherwise the execution will be skipped and the state will be immediately set to *aborted*. The expression returns `False` if there is at least one predecessor which is in state *compensation failed* or if there is at least one predecessor which is in state *aborted* and the *forceExecution* flag has not been set. Otherwise R is `True` and the transaction manager sets the state according to the result of the execution of the transaction. As soon as the new state of the current transaction has been set, all dependent transactions as well as the parent transaction (if existing) will be woken up by setting their state to *waiting for execution*.

Now we take a look at the compensation handling in figure 4.7. If not all compensa-

tion predecessors are in a final state, then the transaction is reset to state *suspended*. If no compensation dependencies have been defined, then the compensations will be executed in the reverse order of their original execution schedule. The results of the predecessors (either *compensated* or *compensation failed*) are not considered in the compensation handling. This is a simpler approach than the dependency handling during the normal execution. Otherwise the compensation is started. The state will then be set according to the result of the compensation (*compensated* or *compensation failed*). Then all successors in state *compensation suspended* as well as the parent transaction are woken up. The workflow in both figures leads to the following behavior:
If there is at least one sub transaction in state *compensation failed*, the transaction manager starts the compensation for the global transaction and the state of the global transaction will be set to *compensation failed*. This behavior keeps the consistency violation as small as possible. If no other transaction is in state *compensation failed*, the application only has to take care of the one transaction in state *compensation failed*. The application is able to query the final states of all transactions of a global transaction.

To preserve the consistency between all transaction entries it is required to perform one instance of a transaction manager's workflow in a single local transaction. If this local transaction is an ACID compliant distributed transaction then

1. the execution space does not need to be the same space as the space of the transaction manager and
2. sub transactions themselves can perform operations on remote spaces with the same transaction which is used by the transaction manager for the update of its internal data structures.

For this purpose the Extended Paxos Commit (see definition in section 4.2) algorithm could be used. One could argue that sub transactions of Extended Paxos Commit are not required, since the REXX model also supports sub transactions. However Extended Paxos Commit commits sub transactions ACID compliant, which is not supported by REXX transactions. So for example ACID compliant function replication could be modelled.

One of the goals of the new REXX transaction model is to support long lived transactions (LLT). The execution of a LLT can take several hours and therefore also the transaction manager is blocking during this time. This blocking state can be circumvented if the Paxos transaction is executed asynchronously. This increases the utilization as well as the throughput of the individual transaction managers. In that scenario the responsibility of the error handling, which is required when the commit fails, is passed to the guard processes which are explained in section 4.3.6. When the asynchronous commit fails, the transaction is stuck in state *running* since it will never be processed by any transaction manager.

Since the transaction manager handles the whole workflow within one single local transaction it will also lock all required resources (dependencies, predecessors) for a long time and will so affect the concurrency. Therefore the workflow of leaf transactions is internally split into two parts with two local transactions. Within the scope of the first transaction all predecessors are checked and the state is set to *running*. The second transaction spans over the execution as well as the update of the internal system states. This approach improves the concurrency but also harms the consistency of the transaction manager. This problem is discussed in the next section.

4.3.6 Orphan Transactions

Before the transaction manager executes a transaction it sets the transaction's state in a separate local transaction (see figure 4.6) to *running*. Therefore if the transaction manager crashes during the execution of a transaction the transaction entry remains in that state. The problem is the same when the transaction manager crashes during the execution of compensation actions.

Imagine several transaction managers are running on different locations and are operating on a replicated shared memory. Then one location suffers a power outage and afterwards all transaction managers are restarted. If transactions (handled by transaction managers which have been run at the crashed location) have been in state *running* during the crash, then they become orphans because no one will take care of them anymore.

The problem here is to detect orphan transactions and abort them appropriately. But when looking at all running transaction entries in the space then it can not be determined whether they are handled by a transaction manager or not. The transaction model supports timeouts, however it specifies neither a limit for the timeout nor how long a transaction can stay in state *running*. This can take seconds, hours or even days.

Nevertheless there are three methods to detect and handle orphan transactions:

- **Keep-alive pings.** During the execution of a transaction, transaction managers periodically write a keep-alive message with the name of the current transaction into a certain place. Whenever new transaction managers start or recover from a failure they first check for all running transactions whether they are alive. If there was no keep-alive message for a transaction within a certain period, the transaction manager will take over responsibility and abort the transaction. The downside of this method is that every transaction manager has to be equipped with a separate thread whose only responsibility is to periodically write keep-alive messages. This results in a high resource consumption and even if the proposed thread is sending its keep-alive messages it can not be guaranteed that the main thread of the transaction manager is still running correctly.

- **Guard processes.** Beside transaction processes, dedicated guard processes are introduced and operate on the same data as transaction processes. Guard pro-

cesses periodically query all running transactions and check their deadline. If the deadline has been exceeded then they will take over responsibility and abort the transaction. The deadline of a transaction is set when it starts. The computation of the deadline is described in section 4.3.7. This either requires synchronized clocks of all transaction managers or the current time is retrieved from the environment of the shared space. In practice it is recommended to consider a buffer in the deadline computation. This will prevent situations where a transaction manager is preempted by a guard process although it would have finished execution in the next seconds. One downside of this method is that long-running orphaned transactions will not be detected before their deadline.

- **Transaction Manager Identifiers.** Every transaction manager has a unique identification number. Whenever it starts the execution of a transaction it also writes its ID into the transaction entry. So every running transaction can be assigned to a transaction manager. For this transaction managers must guarantee that they recover within a certain period after a crash. During the startup the transaction manager looks for a running transaction which is marked with its ID. If it finds an orphan transaction which is labelled with its ID it will abort it appropriately. There can be at most one orphan transaction for a certain transaction manager at the same time. The requirements for this method are very high because the guaranteed period within a crashed transaction manager has to recover depends on the transaction timeouts of the transactions, and transaction timeouts can not be influenced by the transaction system.

It has been decided to use *Guard Processes* for the handling of orphan transactions in the REXX model because

- the implementation of guard processes is simple,
- they only require little resources and
- no assumptions have to be made regarding the recovery of transaction managers.

4.3.7 Timeout Handling

The timeout handling of the REXX model is based on three different types of timeouts:

- The **global timeout** is passed by the application in the `commit` call of the API. This timeout is converted in the transaction manager to an absolute time and acts as the *global deadline*. This deadline overrules the individual timeouts of all leaf transactions if the global deadline is earlier than the deadlines which correspond to the individual timeouts. However the global deadline does not overrule the compensation timeout of transactions.

- For each leaf transaction an **individual timeout** can be set by the application. When the execution of a leaf transaction is not finished within that period, the transaction manager aborts the leaf transaction and performs a rollback. In that case the state of the leaf transaction is set to *aborted*.
- The **compensation timeout** can also be specified for each sub transaction individually by the application. It determines the timeout for the execution of the compensation of a sub transaction. If the execution time exceeds the timeout, then the transaction manager aborts the execution, performs a rollback of the compensation action and sets the state of the sub transaction to *compensation failed*. Since this behavior can cause inconsistency of the data the compensation timeout should be used carefully.

All timeouts have to be specified in advance by the application before the global transaction is passed to the transaction manager. If no timeout value is specified, then an infinite timeout is assumed.

4.3.7.1 Individual transaction timeout

The individual transaction timeout only applies to the execution of transactions and does not affect the execution of compensations. Before the transaction manager starts the execution of a transaction it first determines all timeouts using the following rules:

- If the timeout of a transaction has not been set by the application, then the timeout is set to *infinite*.
- The timeout is set to the minimum of the parent's timeout and the timeout of the transaction. This keeps a hierarchical structure of the timeout values.

Whenever leaf transactions start their execution a deadline value is computed using the formula

$$deadline = min(currentTime + txTimeout, globDeadline) \tag{4.3}$$

where $currentTime$ is the current time, $txTimeout$ is the timeout of the leaf transaction and $globDeadline$ is the global deadline of the global transaction. This calculation assures that

1. the leaf transaction does not take longer than its specified timeout and
2. the leaf transaction does not take longer than the global deadline (if the execution engine correctly handles the timeout).

So both values are considered: the global deadline of the global transaction as well as the relative timeout of the leaf transaction. Even if there are dependencies between transactions, this timeout method assures that the global timeout is always considered.

4.3.7.2 Compensation timeout

The compensation timeout is applied when the compensation of a transaction is started. There are some differences regarding the individual transaction timeouts:

- There is no global deadline for the compensation.
- When the individual timeout elapses the state of the transaction is set to *compensation failed*. This can result in an inconsistent state of the system.
- Compensation timeouts are not inherited to sub transactions.

Compensation timeouts should be used carefully because the system can become inconsistent when compensation actions are aborted. Since all transactions (parents and leaves) have compensation actions, the timeout is not inherited to the sub transactions. Therefore compensation timeouts have to be explicitly specified for each individual transaction.

4.3.7.3 Timeout Handling: Examples

The following examples explain the timeout handling in more detail.

Example 1: Transaction with a global timeout Figure 4.8 shows a global transaction with a global timeout of 20 seconds which consists of the parent transaction A and the two sub transactions B and C. No individual timeouts have been specified and therefore the default timeout (infinite timeout) is assumed. In addition transaction C depends on transaction B.

The following steps are performed:

1. Transaction B first starts the execution. No individual timeout has been set, therefore it is assumed as infinite and the deadline is set to the global deadline (20s) (formula 4.3).
2. Transaction B finishes after 8s and C starts the execution. The deadline is again set to the global deadline (20s) because an individual deadline has been specified.
3. Transaction finishes after 17s.

Example 2: Transaction with a global timeout and individual timeouts Figure 4.9 shows a global transaction with two nesting levels and a global timeout of 20 seconds. Additionally for every sub transaction individual timeouts have been set. Transaction C depends on B and transaction D depends on BC.

The following steps are performed:

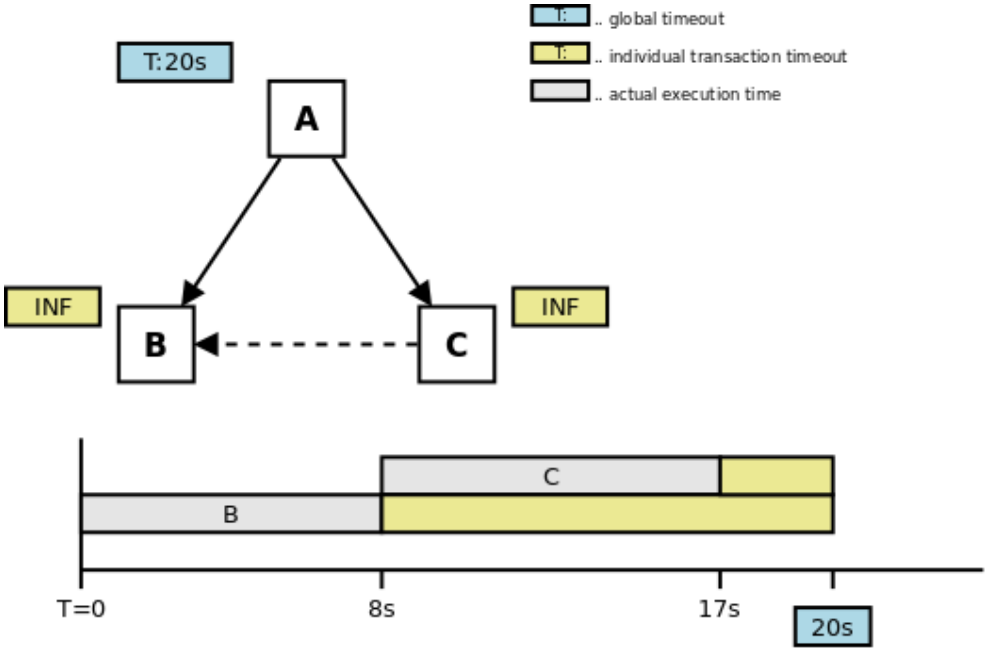

Figure 4.8: Global transaction with a global timeout

1. Transaction B first starts the execution. The actual execution deadline is computed using the formula 4.3 and set to 10s.
2. Transaction B finishes after 8s and C starts the execution. The deadline is set to 13s (8s + 5s).
3. Transaction C finishes after 12s and D starts the execution. The deadline is now set to 20s because the global deadline is lower than the current time plus the individual timeout.
4. Transaction finishes after 18s.

Example 3: Transaction with multiple timeouts Figure 4.10 shows a global transaction with a global timeout of 15 seconds which consists of one parent transaction and two sub transactions. The sub transactions have an individual timeout of 15s and a compensation timeout of 5s. Additionally transaction C depends on B.

The following steps are performed:

1. Transaction B first starts the execution. The actual execution deadline is set to 15s.
2. Transaction B finishes successfully after 12s and C starts the execution. The deadline is set to 15s.

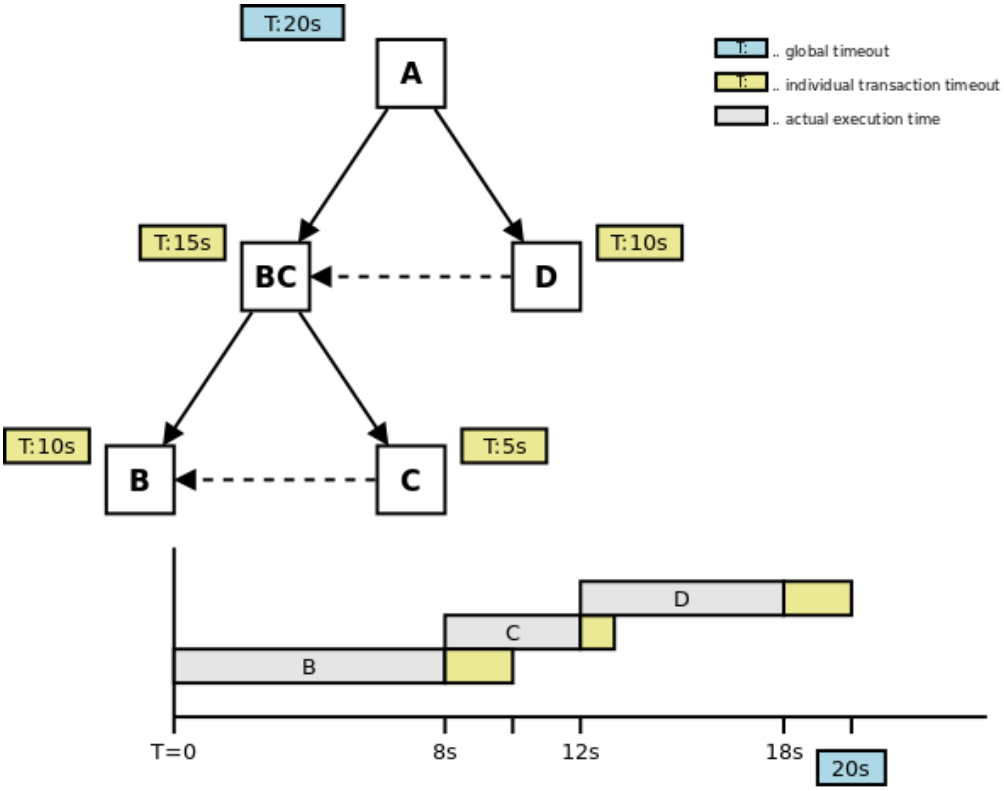

Figure 4.9: Global transaction with a global and individual timeouts

3. Transaction C is aborted by the transaction manager after 15s since its timeout has elapsed. The parent transaction A triggers the compensation of transaction B. The deadline of the compensation is set to 20s (15s + 5s).

4. The compensation successfully finishes after 17s.

If the compensation had not finished before the deadline, then the transaction manager would have set the state to *compensation failed*.

4.3.8 The global context

From the moment when the application has handed over the global transaction to the transaction manager the application can not access the global transaction until it has been completed. So the application can not take care of the coordination between transactions of the global transaction. But there are a lot of cases where communication between transaction is required and therefore the global context provides exactly this possibility.

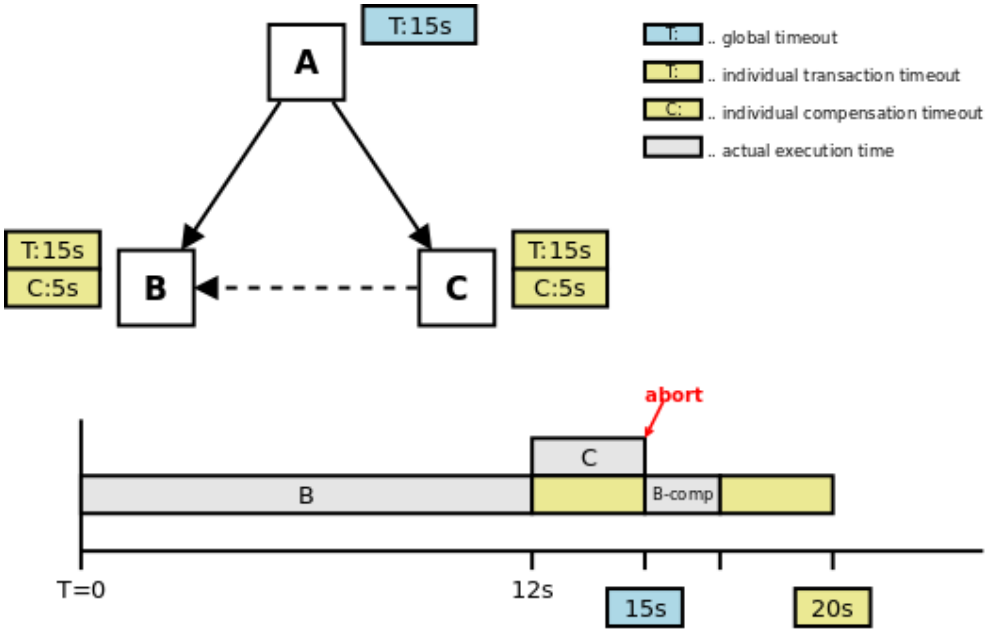

Figure 4.10: Transaction with several timeouts

The global context is a simple key-value store which provides two access methods (`put` and `get`). Each sub transaction has access to its local context as well as to the context of the parent.

The `get`-method first tries to obtain the value from the local context and when it fails it accesses the context of the parent, whereas the `put`-method always writes to the local context so that the sub transaction does not modify the context of the parent during its execution.

After the execution of a transaction has finished the local context is inherited to its parent where entries with the same key will be replaced. Since it is not possible to delete values from the context, it monotonically grows after every transaction hierarchy.

Figure 4.11 shows a 2-level nested transaction. Parent transaction B inherits both context values from its sub transactions D and E. After completion of transaction B its context is passed to its parent A which also inherits the context of C. When the global transaction has been completed all context values are stored in its top-level transaction.

If transaction D wants to pass information to transaction E, then a dependencies has to be added to assure that transaction E is not executed before transaction D has committed. This is required because the context values of transaction D are not visible to other transactions until it has committed. After completion of the global transaction the whole context is passed to the application as part of the result.

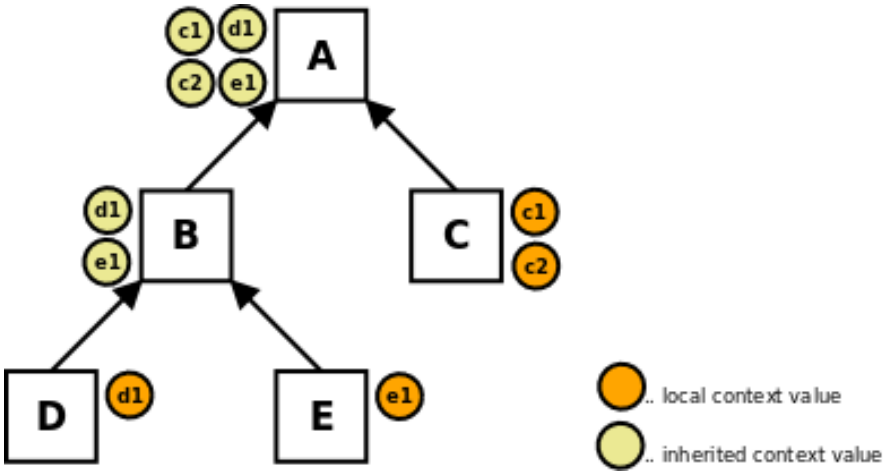

Figure 4.11: Context inheritance between transactions

4.3.9 Scheduling of REXX transactions

The following examples explain how REXX transactions with different structures are processed according to the described model.

4.3.9.1 Example 1: Transaction with XOR

The following example shows how function replication could be modelled with the REXX transaction model. Figure 4.12 shows a global transaction which follows the composition $X = (A \oplus B) \wedge C$. For the modelling of this composition it is required to introduce an intermediate transaction AB which stores the result of $A \oplus B$. The aggregation function of transaction AB is $A \vee B$ whereas transaction X uses the default aggregation function (AND). The figure also shows that transaction B depends on A and A has set the *forceExecution* flag, so that it is executed even if its predecessors have aborted. This dependency already defines the priority of the two transactions. In this case transaction A is executed and its result is passed to transaction B. Transaction B has to evaluate the result of A and has to abort if A has been successful, otherwise B will be executed. This functionality has to be implemented by the developer and is not part of the transaction model.

When the global transaction is passed to the transaction manager it is split into five transactions which are represented by five entries in the data container. All of the transactions are initially set to state *waiting for execution*. Table 4.1 shows one possible execution schedule of a successful execution of the transaction. It lists all states (*(W)aiting*, *(S)uspended*, *(R)unning*, *(C)ommitted*, *(A)borted*) in every round. For the schedule it has been assumed that more than one instance of the transaction manager is running in parallel.

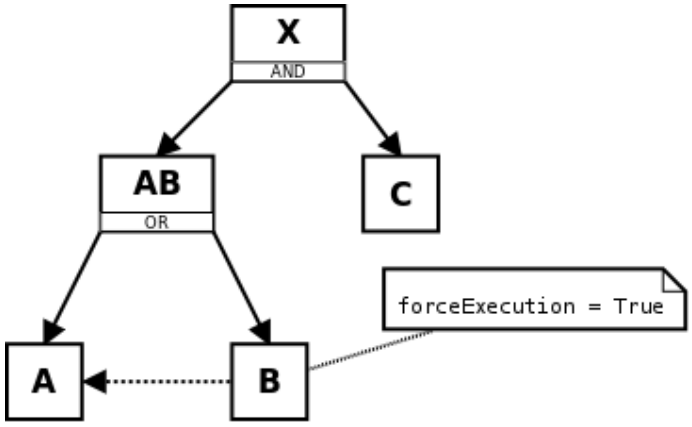

Figure 4.12: Transaction diagram for composition $X = (A \oplus B) \wedge C$

round	A	B	AB	C	X
0	W	W	W	W	W
1	W	S	S	R	S
2	R	S	S	C	W
3	R	S	S	C	S
4	C	W	W	C	S
5	C	R	S	C	S
6	C	A	W	C	S
7	C	A	C	C	W
8	C	A	C	C	C

Table 4.1: Example execution schedule of transaction X

Here is the explanation of the rounds in table 4.1:

1. Since the dependencies are not met, transactions B, AB and X are set to *suspended*. Transaction C has no dependencies and therefore starts the execution. It can be seen that for the execution of this round four transaction managers are working in parallel. Also a fifth instance would be busy if the transition of transaction A to state *running* would be moved to this round.

2. Transaction A starts its execution. Meanwhile transaction C has been committed and the parent transaction X was awakened.

3. Since not all sub transactions of X are in a final state it returns to state *suspended*.

4. Now transaction A successfully completes and awakes the parent AB as well as the successor transaction B.

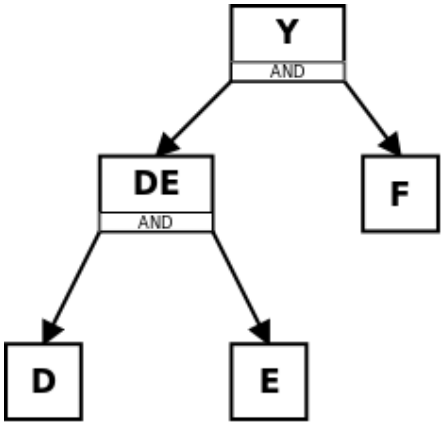

Figure 4.13: Transaction diagram for composition $Y = (D \wedge E) \wedge F$

5. Transaction B starts its execution and AB is set to *suspended*.

6. Since A was successful the execution of transaction B returns *aborted* (This has to be assured by the developer). This causes transaction AB to awake.

7. Now all dependencies of AB are met because all sub transactions are in a final state. The aggregation function $(A \vee B)$ evaluates to true and the state is set to *committed*. In the same instance the parent transaction X is set to *waiting for execution*.

8. Transaction X evaluates its aggregation function (AND) and finishes the global transaction in state *committed*.

4.3.9.2 Example 2: Failed Compensation

This example shows a two-level nested transaction with the composition $Y = (D \wedge E) \wedge F$ (see figure 4.13). Both parent transactions (DE and Y) derive their result by performing the default AND aggregation function. There are no dependencies between sub transactions.

Again the transaction manager splits the transaction into five transactions and sets them initially into the *waiting for execution* state. Table 4.2 now shows one possible execution schedule for the scenario that transaction E aborts and subsequently the compensation of the already committed transaction D fails. The table now uses four more states: *waiting for compensation (WC)*, *running compensation (RC)*, *compensated (COMP)* and *compensation failed (F)*.

Here is the explanation of the rounds in table 4.2:

round	D	E	DE	F	Y
0	W	W	W	W	W
1	R	R	S	R	S
2	A	R	W	C	W
3	A	R	S	C	S
4	A	C	W	C	S
5	A	WC	S	C	S
6	A	RC	S	C	S
7	A	F	W	C	S
8	A	F	F	C	W
9	A	F	F	WC	S
10	A	F	F	RC	S
11	A	F	F	COMP	W
12	A	F	F	COMP	F

Table 4.2: Example execution schedule of transaction Y

1. Transactions D, E and F are starting execution while the parent transactions are set to *suspended*. For this schedule we assume that there are at least five active transaction manager threads.

2. The execution of transaction D aborts and DE is set to *waiting for execution* again, whereas transaction F has committed successfully and its parent Y is set to *waiting for execution*.

3. Transaction DE is set to *suspended* again because not all sub transactions are in a final state. The same is valid for Y.

4. Now transaction E commits and sets DE to *waiting for execution*.

5. Transaction DE executes its aggregation function which yields *false*. Therefore the compensation of E is triggered.

6. Compensation of E is running.

7. Compensation of E has *failed* and DE is set to *waiting for execution*.

8. Transaction DE inherits the state *compensation failed* and then sets its parent Y to *waiting for execution*.

9. Top-level transaction Y then triggers the compensation of the already committed transaction F and waits for it in state *suspended*.

10. Compensation of F is running.

11. Compensation completes and therefore Y awakes.

12. Transaction Y inherits the state *compensation failed* and the transaction workflow is completed.

Whenever a global transaction in state *compensation failed* is passed back to the application the application is responsible to restore a consistent state.

4.3.10 Proof of consistency

For the proof of consistency the following invariants have been checked with a model checker:

- If a parent transaction is in a final state, then all of its sub transactions must be in a final state.
- If a sub transaction is in state *compensation failed* and the parent transaction is in a final state then the parents state must be *compensation failed*.
- If a parent transaction is in state *committed* then at least one sub transaction must be in state *committed*.

For the full proof regarding consistency TLA$^+$ (The Temporal Logic of Action Plus) [Lam02] has been used for the formal specification of the algorithm (see appendix A). Afterwards the specification has been checked by using TLC. TLC is the model checker which is part of the TLA$^+$ toolbox. TLA has been designed for specifying concurrent system by using mainly ordinary (non-temporal) mathematics. TLA$^+$ is based on TLA and is a complete formal language. It includes ordinary math like first order logic and sets. In addition it provides constructs for writing proofs.

CHAPTER 5

Implementation

This chapter describes how the two relaxed transaction models have been integrated into the current version of MozartSpaces. The goal was to use as many features of MozartSpaces as possible to yield optimal performance. The big challenge is to align the architecture of the implementation to the architecture of MozartSpaces. This means to have a data-driven approach where the data controls the coordination between all processes.

Throughout the implementation the focus has been set on scalability. This requires that

- all processes can operate concurrently on the data and
- the dependencies between the data are as low as possible.

5.1 Extended Paxos Commit

5.1.1 Architecture Overview

Like any other extension of MozartSpaces the implementation of the extended Paxos Commit algorithm has to be totally transparent to the application. The application should only see the additional methods which are provided through the API. The whole protocol has been implemented on top of MozartSpaces. Figure 5.1 shows that an additional Paxos layer has been introduced between the MozartSpaces core and the API. The Paxos layer operates like any other application and it also uses the API to access the core. This design keeps a clear interface between the MozartSpaces core and the new Paxos implementation. However minor changes within the core are necessary.

Based on the current MozartSpaces architecture the changes of the new extension are performed in the following areas:

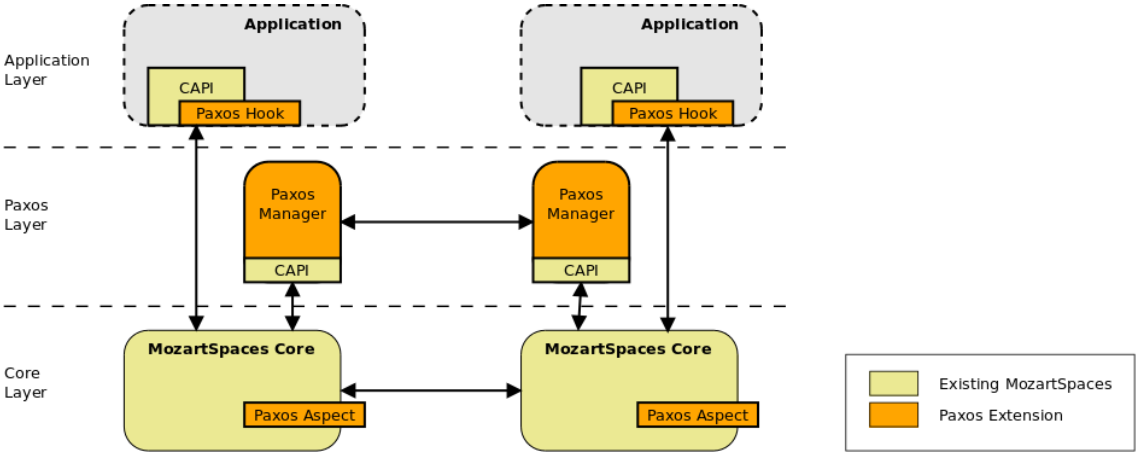

Figure 5.1: Architecture of MozartSpaces with the new Paxos Extension

- Extension of the CAPI - new methods are added for the Paxos implementation
- The Paxos Manager - main part of the extension
- Paxos SpaceAspect - guarantees consistency after restart
- Minor changes in the MozartSpaces core

5.1.2 Extension of the CAPI

The CAPI is the API which is used by applications to communicate with MozartSpaces.

Transactions in MozartSpaces are identified by the URI of the space where the transaction has been created as well as an ID. Both values together form the unique `TransactionReference`. The new `PaxosTransactionReference` extends the existing transaction reference by the values listed in table 5.1:

The transaction reference stores all data which is required to perform the execution of the Paxos Commit algorithm.

The existing `CAPI` has been extended and now provides additional methods. Table 5.2 lists some of the new methods of the class `PaxosCapi`. Further methods are provided but they only differ in their arguments and not in their functionality.

Table 5.3 lists the values of the enumeration `PaxosStatus`. It contains all possible states of the Paxos protocol. For the application only `COMMIT` and `ABORTED` are visible. The other states are only used internally.

The main task of the `createTransaction()` methods is to obtain a global transaction ID. In the original implementation this call receives a new transaction ID from the core. But since the existing persistence implementation does not include the counter of the transaction ID, the transaction ID is reset after a restart of the core. For

Attribute	Description
`URI registrar`	URI of the space where the registrar process is located
`ArrayList<URI> acceptors`	list of spaces where the acceptor processes are located
`Long timeout`	timeout of the transaction
`ArrayList <PaxosTransactionReference> lstSubTx`	list of all sub transactions
`PaxosTransactionReference parent`	parent transaction
`distributedFlag`	indicates whether the transaction is a local or a distributed transaction
`ArrayList<URI> participants`	list of participants (resource managers) of the transaction

Table 5.1: The `PaxosTransactionReference` class

distributed transactions it is essential to have unique transaction IDs. This has been accomplished with a persistent container which stores the value of the next ID. The implementation of Paxos completely encapsulates the usage of local transactions. Local transactions can not be directly created using the `PaxosCapi`. The new CAPI always returns a `PaxosTransactionReference`, however if only one space is involved in a Paxos transaction, the local transaction system is used for the commit.

The `commitTransaction()` methods trigger the distributed commit algorithm. Instead of calling the local commit function, a message is sent to the registrar process. This message starts the distributed transaction. The synchronous call is blocked until the result of the transaction is received from the leader process of the transaction. Because of the asynchronous characteristic of the Paxos protocol it can not be guaranteed that all participating nodes already have committed their local data after this method returns.

All other transactional operations of the original `CAPI` like `write()` or `take()` have been extended to

- add the target space of the operation to the list of participating spaces in the `PaxosTransactionReference` and
- set the flag `distributed` of `PaxosTransactionReference` when the target space of the operation is not the local space.

Method	Description
`createTransaction( timeout, space)`	returns a new `PaxosTransactionReference` with the next transaction ID of space `space` and the specified timeout.
`createTransaction( timeout , space, context, acceptors)`	returns a new `PaxosTransactionReference` with the given values. The default list of acceptors is replaced by `acceptors`.
`createTransaction( timeout, context, parent)`	creates a new sub transaction of `parent`. The new `PaxosTransactionReference` is added to the reference of the parent.
`commitTransaction( transaction, rule, context)`	starts the Paxos Commit algorithm for the given `transaction` and the commit rule `rule` and returns the `PaxosStatus`.
`commitTransaction( transaction)`	starts the Paxos Commit algorithm for the given `transaction` and the default commit rule and returns the `PaxosStatus`.
`commitTransactionAsync (transaction, context)`	starts the Paxos Commit algorithm for the given `transaction` with the default commit rule and returns immediately.
`rollbackTransaction( transaction)`	starts the Paxos Commit algorithm for the given `transaction` and forces `ABORT` for the result.

Table 5.2: The `PaxosCapi` class

5.1.3 Paxos Integration

The Paxos protocol specifies different roles and each of these roles is implemented as a single class which extends the `Thread` class.

5.1.3.1 Communication between processes

The communication and coordination between the Paxos processes is accomplished with MozartSpaces. Every space has its own message container. If a process located on space A wants to send a message to a process on space B then it writes a message into the message container of space B. Table 5.4 gives an overview of all used containers.

The message container is non-persistent and is created by the Paxos manager before the startup of the other Paxos processes. The decision to use a non-persistent container is based on the fact that operations on non-persistent containers are faster. Nevertheless if a space restarts and the container is destroyed the operation is not affected since the Paxos model can cope with message loss. Every process uses two containers, the

Status	Description
WORKING	the transaction has been used but the commit cycle has not started yet. This state is only used internally.
PREPARED	the transaction has already been prepared by the resource manager. This state is only used internally.
COMMIT	the transaction has been committed.
ABORTED	the transaction has been aborted and a rollback has been performed.

Table 5.3: The PaxosStatus enumeration

Container name	Persistent	Description
__paxos_metadata	Yes	stores the next transaction ID.
__paxos_msg	No	used for the communication between Paxos processes.
__paxos_log	Yes	used to keep prepared data persistent. This container is required to prevent inconsistent system states after a system crash.
__leader_election	No	used for the communication between leader processes.
__paxos_rm_data	Yes	stores the operational data of the resource manager process.
__paxos_reg_data	Yes	stores the operational data of the registrar process.
__paxos_acc_data	Yes	stores the operational data of the acceptor process.
__paxos_lead_data	Yes	stores the operational data of the leader process.
__paxos_prop_data	Yes	stores the operational data of the proposer process.

Table 5.4: List of containers used in the Paxos Implementation

messages container as well as a separate persistent data container which acts as the stable storage in the Paxos Commit protocol.

The message container is coordinated with a LabelCoordinator whereas the persistent data containers are using a KeyCoordinator with the transaction reference used as key.

5.1.3.2 The `PaxosManager`

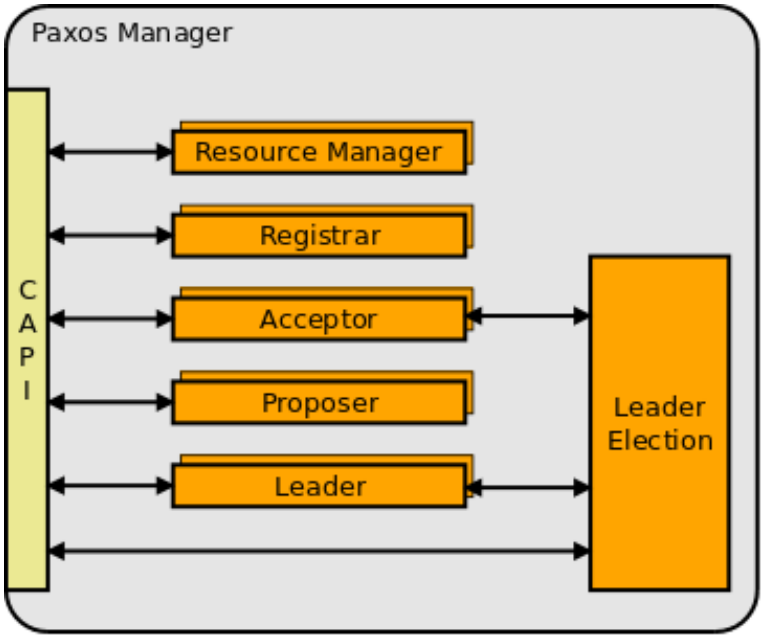

Figure 5.2: Architecture of the Paxos Manager

The Paxos Manager is the main part of the implementation and it is started after the initialization of the MozartSpaces core. It behaves like a normal application which is communicating via the standard CAPI with the MozartSpaces core. It is responsible for the following operations:

- Creation of the message container
- Creation and startup of Paxos processes
- Creation and startup of Paxos leader election process
- Setup of the Paxos aspect
- Shutdown of all Paxos processes

5.1.3.3 Scalability

Since all processes, regardless whether they reside on the local space or not, only communicate with the space, it is possible to run several processes in parallel without additional coordination effort. So depending on the need there can run one, two or more instances of the same Paxos process on one space. They all share the same persistent data container.

5.1.3.4 Space Aspects

For the implementation of the distributed commit algorithm it is required to modify the start-up behavior of the space. Due to the flexibility of MozartSpaces those changes can be incorporated into an aspect. The goal of the aspect is to prevent inconsistent system states after a restart of the space. There are two cases which can result in an inconsistent system state:

1. The space performs a transactional operation like `take()` or `write()`. Afterwards due to some reason the space crashes. Since the current implementation of the persistency does not include the isolation manager (see section 3.2.2), the operation is lost after the space has recovered. However the application thinks that the call has been successful and so the application data might become inconsistent if the subsequent commit succeeds.

2. The transaction has already been `prepared` and the space assured to be able to commit at any time. If the space now restarts the prepared data is lost. This is because the current implementation of the persistency does not include prepared transactions. If the outcome of the distributed transaction is `commit`, the application data becomes inconsistent.

The first case is handled by the aspect `PaxosSpaceAspect` which is plugged to the interception point (`IPoint`) before all transactional operations (`ALL_PRE_POINTS`). The aspect then creates a persistent entry for each transaction in the data container of the resource manager process of the local Paxos instance. If an entry of the transaction already exists no action will be performed. The second part is to mark all transactions as `invalid` after the startup of the space. This is performed in the startup phase of the space when the space aspect is added. This causes all subsequent *prepare* operations to fail.

Due to the current architecture of MozartSpaces and the current persistency implementation it is not possible to guarantee a consistent state in the second case. Once a space has communicated *prepared* to the Paxos protocol it can not withdraw its decision. It has assured that it is possible to *commit* at any time. This promise only holds if the system is able to store the *prepared* data persistently so that it survives a restart of the space. In the current implementation this can not be managed without spending a tremendous effort. So another approach has been chosen to cope with that case.

During the startup phase of the space all prepared transactions are marked as *invalid*. Once a transaction is prepared, there are only two possible options: *commit* and *abort*. In the *abort* case we are happy because we do not have to perform special actions since the prepared data has already been lost after the restart of the space. In the `commit` case the validity of the transaction is checked. If it is marked as *invalid* then a `ContainerAspect` is created for each local container which has been affected by

the transaction. The only job of this aspect is to raise an `InconsistentContainer` exception whenever a process tries to access the container. Since aspects are not covered by the current implementation of the persistency profile, the aspect has to be re-created after every restart. Once a container is marked as *inconsistent* the application is responsible to recover it.

5.1.3.5 Changes in the MozartSpaces core

1. The current implementation does not allow operations which are performed with a transaction from a remote space. This constraint has been removed.

2. If an operation with a new transaction is executed, the new transaction will be implicitly created. Therefore the passed Paxos transaction reference is used as the local transaction reference. So all cores which have joined in the same distributed transactions share the same transaction reference. The current implementation throws an error if a transaction has not been created before its first use. The new approach was necessary because until a MozartSpaces core receives the first operation of a distributed transaction it is not clear whether the core joins the transaction or not. Another approach would be the explicit creation of a local transaction just before the first operation of a distributed transaction is executed on the core. In that case a modification of the `createTransaction` method would have been required so that it can handle already existing transaction references.

3. After the resource manager has prepared a transaction, it is not allowed to execute further operations with that transaction. The `status` attribute of `DefaultTransaction` is set to `COMMITTING` when the transaction has been prepared. So the MozartSpaces core does not allow further operations. The attribute is set in the aspect `PaxosSpaceAspect`.

4. Currently MozartSpaces handles a timeout for every transaction and if this timeout elapses a rollback is performed. But since already prepared transactions must not time out, the `TransactionTimeoutHandler` has been adapted accordingly.

5.1.3.6 Configuration

The configuration of the Paxos protocol is done in the class `PaxosConfiguration`, which is part of the MozartSpaces configuration `CommonsXmlConfiguration`. The configuration can be specified either directly in the application or through the XML configuration file. The configuration contains the following values:

- **Phase3 timeout** - The timeout in the resource manager specifies the period after which a re-send of the phase 3 message is requested. Default value is 3 seconds.

```
<mozartspacesCoreConfig>
    <!-- ... normal MozartSpaces config is here ... -->
    <paxos>
        <phase3Timeout>3000</phase3Timeout>
        <leaderTimeout>6000</leaderTimeout>
        <nbrHandlersRM>2</nbrHandlersRM>
        <nbrHandlersREG>1</nbrHandlersREG>
        <nbrHandlersACC>2</nbrHandlersACC>
        <nbrHandlersPROP>1</nbrHandlersPROP>
        <nbrHandlersLEAD>1</nbrHandlersLEAD>
        <acceptors>
            <acceptor>xvsm://localhost:9876</acceptor>
            <acceptor>xvsm://localhost:9877</acceptor>
        </acceptors>
    </paxos>
</mozartspacesCoreConfig>
```

Listing 5.1: Sample Paxos configuration

- **Leader timeout** - This timeout is related to the leader election service. If there has not been a signal from the current leader within that timeout, a new leader will be elected. The default value is 6 seconds.
- **Default acceptors** - This is the list of acceptor nodes (specified by their URI), which is used by Paxos if no acceptors have been specified in the commit method. If the set of acceptors is neither specified in the configuration nor in the commit call, then the current core is used as the only acceptor.
- **Number of processes** - For all Paxos processes the number of running threads can be specified individually. Per default there is one thread running for each Paxos process.

Listing 5.1 shows a sample XML configuration for Paxos.

5.1.4 Paxos Processes

The abstract class `PaxosHandler` acts as framework for all Paxos processes and handles the communication as well as the persistent data. The class is generic and must be parameterized with a class of type `PaxosData`. The class `PaxosData` itself is an abstract class and is used to store the data of one transaction persistently. Every Paxos process has its own implementation of `PaxosData` and `PaxosHandler`. The relation between the classes is shown in figure 5.3.

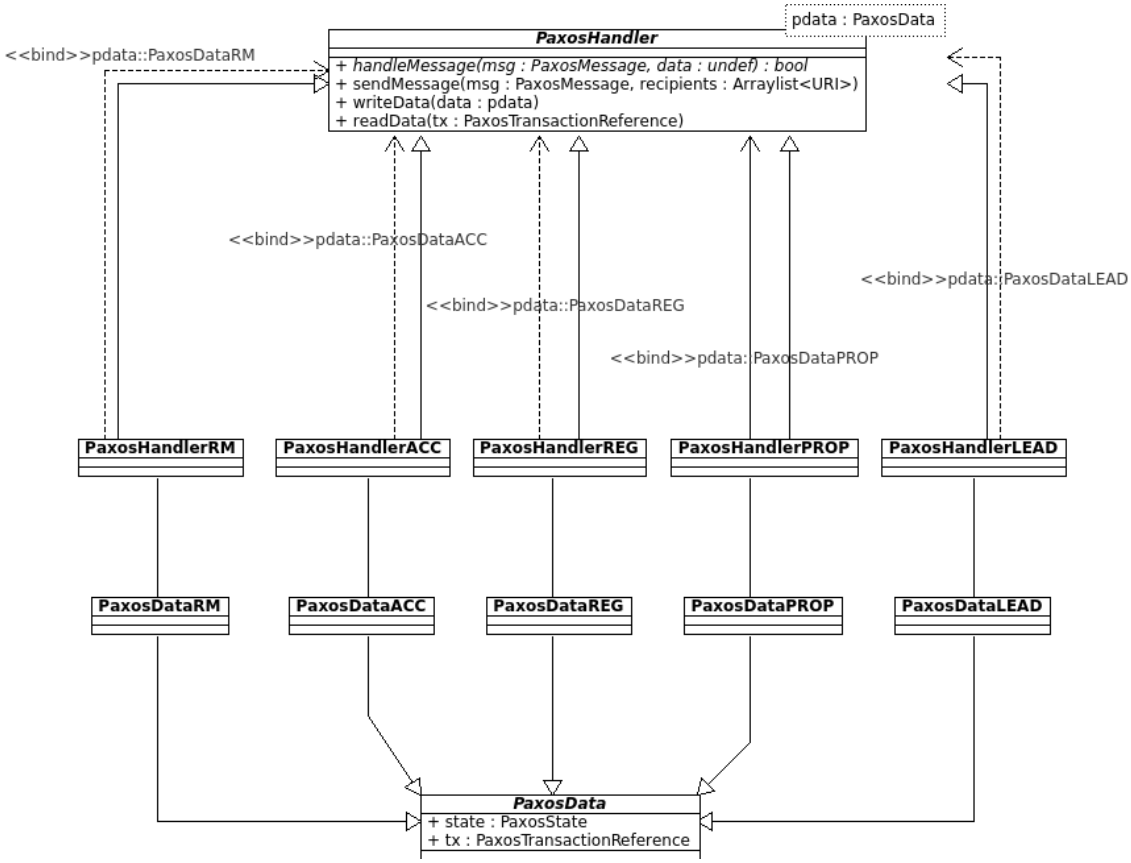

Figure 5.3: Class Diagram of the Paxos processes

The class `PaxosHandler` extends the `Thread` class and so all instances are running in a separate thread. The `PaxosHandler` performs the following workflow in an endless loop:

1. Create a new local transaction.
2. Wait until a new message has been received and consume it.
3. Take the data of type `PaxosData` which is related to the transaction reference of the received message.
4. Call the abstract method `handleMessage`. The task of this method is to put subsequent messages into the outbox and to modify the state of the transaction which is stored in `PaxosData`. The outbox is a simple internal memory provided by `PaxosHandler`. This buffer for the messages is required, since the messages must not be sent before the local commit has been finished successfully.

5. Write the persistent `PaxosData`.
6. Commit the local transaction.
7. Send all messages which are in the outbox.

Messages are sent by writing the message into the `__paxos_msg` container of the remote space. The sending of messages is performed outside the scope of the local transaction because a local transaction per definition can only operate on a single space.

5.1.5 The Resource Manager Process (`PaxosHandlerRM`)

The resource manager is the link between the Paxos protocol and the local MozartSpaces core. Table 5.5 lists all messages which are processed by the resource manager and also describes subsequent actions.

Message	Action	Next State	Subsequent messages
`PREPARE`	all local changes are prepared by calling `prepareTransaction`	`PREPARED`	`PHASE2A` to all acceptors
`PHASE3`	local commit (if the result is *commit*) or local rollback (otherwise) is performed	`COMMIT` or `ABORTED`	-
`CHECK_RM`	-	-	`OUTCOME (RM)` to the next acceptor in the list

Table 5.5: List of messages which are processed by the resource manager

As soon as the `PREPARE` message is received an internal timer periodically checks whether the result of the transaction has already been received. If not, it sends the message `CHECK_RM` to itself which subsequently queries the state of the transaction from the acceptors. This approach assures a clear and consistent asynchronous behavior.

5.1.6 The Registrar Process (`PaxosHandlerREG`)

This process starts the Paxos Commit protocol after it has received the message `START_COMMIT`. Table 5.6 shows further details.

5.1.7 The Acceptor Process (`PaxosHandlerACC`)

This process acts as the transaction manager and is the most complex Paxos process. Table 5.7 lists all messages which are processed.

Message	Action	Next State	Subsequent messages
START_ COMMIT	-	PREPARED	PHASE2A to all acceptors, PREPARE to all participants

Table 5.6: List of messages which are processed by the registrar

Message	Action	Next State	Subsequent messages
OUTCOME (RM)	if the result is available, the corresponding RM is notified	-	PHASE3 to the corresponding RM
OUTCOME	if the result is not available, the abort is triggered	-	PROPOSE to the local proposer
OUTCOME	if the result is available, it is re-sent to the leader	-	PHASE2B to the leader
PHASE1A	either returns 'free' or the chosen value	-	PHASE1B to sending proposer
PHASE2A	wait for other PHASE2A messages if they are still missing	WORKING	-
PHASE2A	notify leader, if all required PHASE2A messages have been received	PREPARED or ABORTED	PHASE2B to the leader
PHASE3	-	COMMIT or ABORTED	-

Table 5.7: List of messages which are processed by the acceptor

Similar to the resource manager the acceptor also starts an internal timer for each transaction. If the timeout elapses it sends an OUTCOME message to itself to trigger further actions. The message OUTCOME (RM) is sent by resource managers which want to know the result of the Paxos protocol. In contrast to the resource manager, the acceptor uses a randomized timeout value to assure that the PROPOSE message is not sent simultaneously by all acceptors. Otherwise the protocol could get stuck if several proposers are struggling for the highest ballot number.

5.1.8 The Proposer Process (`PaxosHandlerPROP`)

The proposer is notified by the acceptor when a transaction has exceeded its timeout. The proposer then starts a new Paxos ballot where it tries to abort the transaction. Table 5.8 lists all actions which are performed after the corresponding message has been received.

Message	Action	Next State	Subsequent messages
PROPOSE	increment the ballot number of the desired Paxos instance and propose the value ABORTED	-	PHASE1A to all acceptors
PHASE1B	votes for ABORTED (if instance is free) or for the already chosen value (otherwise)	-	PHASE2A to all acceptors

Table 5.8: List of messages which are processed by the proposer

This process does not require internal timeout handling. If it does not receive an answer for the PHASE1A message from the acceptors, it waits until it gets re-triggered by a PROPOSE message from any of the acceptors. The proposer is not part of the original Paxos Commit. In Paxos Commit the operations of the proposer are performed by the acceptors. The usage of a separate proposer process keeps the complexity of the acceptor process on a manageable level.

5.1.9 The Leader Process (`PaxosHandlerLEAD`)

Table 5.9 describes the actions of the leader process.

Message	Action	Next State	Subsequent messages
PHASE2B	notifies all acceptors and participating RMs if a majority of PHASE2B messages has been received and if no parent transaction exists	COMMIT or ABORTED	PHASE3 to all acceptors and RMs
PHASE2B	notifies the leader (itself) if a majority of PHASE2B messages has been received and the transaction does have a parent transaction	PREPARED or ABORTED	PHASE2B to the leader (itself)

Table 5.9: List of messages which are processed by the leader

If a majority of PHASE2B messages has been received for a sub transaction, the leader re-triggers the evaluation of the parent transaction by sending an empty PHASE2B message to itself. The state of the sub transaction then either remains in state PREPARED

(if all PHASE2A messages had been PREPARED) or ABORTED (otherwise) until a final decision has been made by the parent transaction.

The leader process implements the new extension of the Extended Paxos Commit model of section 4.2. It executes the Commit Rule which assigns the final decisions (*commit* or *abort*) to the parent transaction as well as to all sub transactions. All commit rules are implementing the ICommitRule interface which only consists of the method getResult(). This method is called by PaxosHandlerLEAD when the original Paxos Commit instances of all sub transactions have finished their second phase (figure 2.4). In that state the state of the sub transactions is either prepared or aborted. The argument of the method is a map HashMap<PaxosTransactionReference, PaxosStatus> which contains the state of all sub transactions.

The return value contains the following values which are packaged into the class RuleResult:

- The PaxosStatus of the parent transaction (*commit* or *abort*).
- The PaxosStatus of all sub transactions (*commit* or *abort*).

The leader process then communicates the results to

- all resource managers PaxosHandlerRM,
- all acceptor nodes PaxosHandlerACC and
- to the calling CAPI which then passes the result to the application.

The three different commit rules which have been defined in the model of the Extended Paxos Commit have been implemented with the following classes:

- DefaultCommitRule
- ThresholdCommitRule
- LogicalCommitRule

Whereas the implementation of DefaultCommitRule and ThresholdCommitRule has been straight forward, the implementation of LogicalCommitRule was a bit more tricky.

5.1.9.1 The LogicalCommitRule

As specified in the model (section 4.2.4.1) the result of this rule is not only the PaxosStatus of the parent transaction. The logical commit rule also modifies the state of the sub transactions if this is required to commit the parent transaction.

For the implementation of that desired behavior a SAT solver has been used. Other than a simple solver, which just returns whether a boolean equation is satisfiable or not, an enhanced solver was required which also returns a model which solves the equation. Due to its flexibility and its licence (GNU LGPL license) the Java-based SAT solver Sat4J [1] has been used.

The boolean equation of the `LogicalCommitRule` has to be specified by the application in CNF (Conjunctive normal form) form. A clause is a list of literals where a literal is either `POS(tx)` (positive literal) or `NEG(tx)` (negative literal) and `tx` is of type `PaxosTransactionReference` and specifies a sub transaction.

The following values are passed to the SAT solver:

- all clauses specified by the application,
- additional clauses with the single literal `NEG(tx)` for all sub transactions in state *aborted* and
- an ordered list of type `PaxosTransactionReference` which specifies the priorities among the sub transactions.

The solver then returns two values:

1. `True` (if the equation is satisfiable) or `False` (otherwise)
2. the model (the list of assignments for all literals) which solves the equation if the first result is `True`. The returned model is the optimal solution in terms of containing the maximal number of positive literals (`POS(tx)`). If there are several possibilities with the same number of positive literals, then the priorities come into play.

The result is then converted into an `RuleResult` object and is returned to the leader process.

5.1.9.2 Committing a transaction

When the application calls `commitTransaction()` it is checked whether the transaction only operates on the local space. If yes, the Paxos algorithm is skipped and the local commit process is started. If the transaction spans over more than one node then the Paxos algorithm has to be executed.

Figure 5.4 shows the message flow of an error-free instance of the distributed commit algorithm with two resource managers and one acceptor process.

[1] `http://www.sat4j.org/`

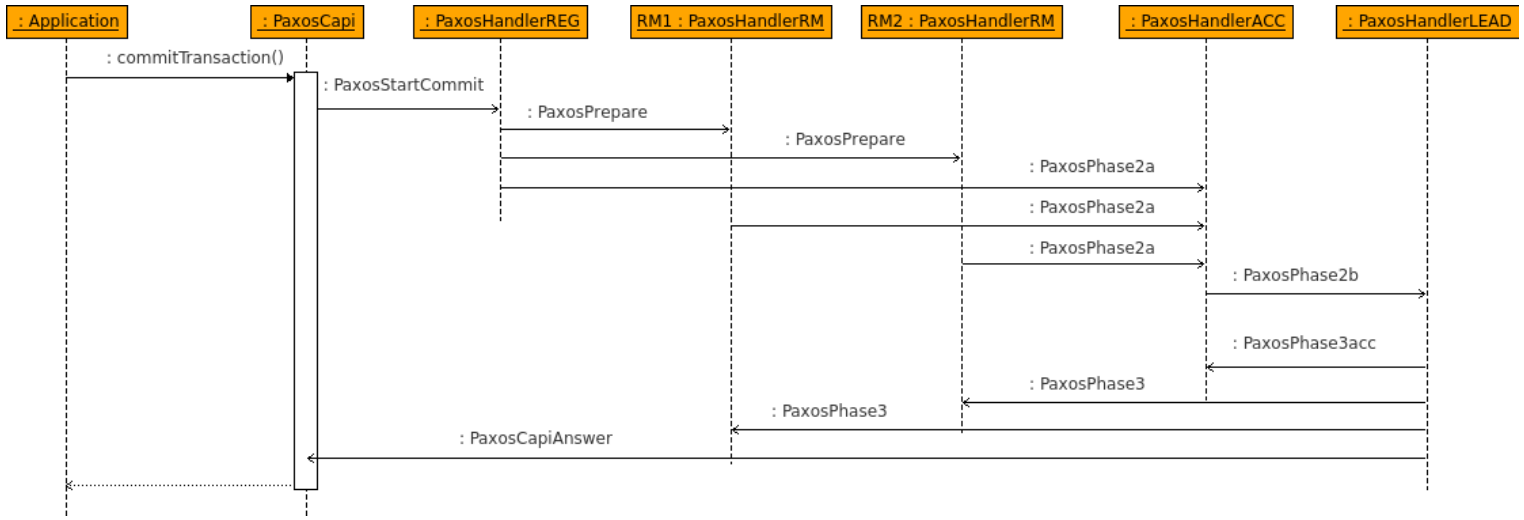

Figure 5.4: Transaction commit

In the 3rd phase the current leader notifies all acceptors, all involved resource managers as well as the calling CAPI. In the synchronous `commitTransaction` operation the call blocks until the outcome (message `PaxosCapiAnswer`) is received. Since Paxos Commit is an asynchronous protocol, we can not rely on how long the transmission of a message takes. So it is possible that the `PaxosCapiAnswer` arrives before the `PaxosPhase3` messages. Then the application thinks that the commit has been processed although the resource managers have not yet started their local commit process. The application only knows that eventually all resource managers will have successfully performed the local commit. The sequence diagram of the CAPI method `rollbackTransaction()` is similar.
One solution to this problem would be the use of notifications. Applications could setup notifications if they necessarily need to know whether a certain entry has been actually written and committed. Another possibility would be to put this kind of notifications within the Paxos implementation, but this would further increase the complexity.

5.1.9.3 The Leader Election

The leader election among all Paxos nodes is performed with a modified version of the stable algorithm described by Marcos K. Aguilera et al. [ADGFT01]. Every node which is running an acceptor process is also running a leader process and the other way round. Every distributed transaction requires a pool of acceptor nodes which can either be specified in the space configuration or can be passed in the commit operation.

Since every transaction can have a different pool of acceptor nodes, there are different approaches for the implementation of a leader election service:

1. Every transaction runs its own instance of the leader election. The advantage of this approach is its flexibility regarding the timeout requirements of the individual transactions. However this causes a high communication effort.

2. Every node runs its own instance of the leader election. This would require that the pool of acceptor nodes is predefined and all transactions are using the same processes. This approach is more efficient, but also limits the flexibility for applications.

3. Every pool of acceptor nodes runs its own instance of the leader election. This is a mix of option 1 and 2.

The third option combines the advantages of the first two options: it is resource efficient and keeps the flexibility of the algorithm. All transactions with the same pool of acceptor nodes share the same instance of the leader election service. All pools are identified with a *group ID* which is a hash value of all nodes in the pool.

A new instance of the leader election service is started when an acceptor receives a `PaxosTransactionReference` with a new *group ID*. The first leader is always the first node of the pool.

5.2 The REXX Transaction Model

5.2.1 Architecture Overview

The implementation of REXX transactions follows a totally different architecture design than the Paxos implementation. The operations of the transactions as well as the coordination among the operations is swapped out into a separate Java class (`RexxTransaction`). This class contains all information which is required to execute the transaction on a separated execution environment, the transaction handler.

Table 5.10 lists all attributes of the `RexxTransaction` class.

The operations of transactions are encapsulated in separate Java classes which have to implement the interface `RexxExecutionCode` which consists of the following two methods:

- `execute(capi, tx, context)`: This method contains the functional logic of the transaction.
- `compensate(capi, tx, context)`: This method is called when the transaction has already committed and the parent transaction decides to compensate the transaction. This method should semantically undo the changes of the `execute()` method.

The access from within those functions is limited to the following objects:

- The `capi` parameter must be used to execute operations on the space.
- The parameter `tx` is the transaction which should be used within the execution body. The timeout of this transaction is based on the `timeout` attribute of the `RexxTransaction`.
- The parameter `context` allows the communication between sub transactions. The context is implemented with the class `RexxContext` which acts as a simple `HashMap` with the signature `<String, Object>`.

The implementation of the timeout of REXX transactions is implemented using the timeout functionality of MozartSpaces. If the timeout of a transaction is detected then MozartSpaces executes a rollback, marks the transaction as invalid and throws an exception. This results in the fact that a correct timeout handling of REXX transactions can only be guaranteed if the execution code keeps the following guidelines:

- No blocking functions (e.g. `sleep()`), except space operations, may be used.
- The surrounding transaction `tx` must be used for all space operations.
- The timeout exception of the surrounding transaction should not be caught within the execution code.

The coordination and communication between applications and transaction handlers as well as the coordination among transaction handlers is accomplished with the special containers listed in table 5.11. All containers are persistent to assure consistency even after a system crash. The containers `__rexx_data` and `__rexx_dependencies` are coordinated with a `QueryCoordinator` because selections with multiple parameters are used. The containers `__rexx_meta` and `__rexx_context` are using a `KeyCoordinator` where the latter additionally uses a `LabelCoordinator`.

5.2.2 Extension of the CAPI

The application interface has been extended by the four methods listed in table 5.12.

The `commitTransction()` method takes a `RexxTransaction` object as argument. The main task of the method is to prepare the transaction so that it can be processed by the transaction handlers without additional effort. Therefore the method recursively loops over all sub transactions of type `RexxTransaction` and

- creates a new entry of type `RexxTransactionEntry`,
- obtains the next unique REXX transaction ID from container `__rexx_meta`,
- sets the timeout to value $min(pTimeout, timeout)$ (where $pTimeout$ is the timeout of the parent and $timeout$ is the timeout of the transaction),
- writes the created `RexxTransactionEntry` into container `__rexx_data` and
- creates a new object of type `RexxDependencyEntry` for every predecessor and writes them into container `__rexx_dependencies`.

These steps are performed in one local transaction. Figure 5.5 shows how the CAPI splits the `RexxTransaction` *tx* into entries of type `RexxTransactionEntry` (*main-tx*, *sub-tx1*, *sub-tx2*). Afterwards the dependencies are written into container `__rexx_dependencies`. The next `read()` operation waits until the top transaction (*main-tx*) has been completed and then collects the result of all sub transactions and also queries the context values.

The methods `commitTransaction()` and `waitForTransaction()` return the result in a `RexxResult` object. Table 5.13 lists all attributes.

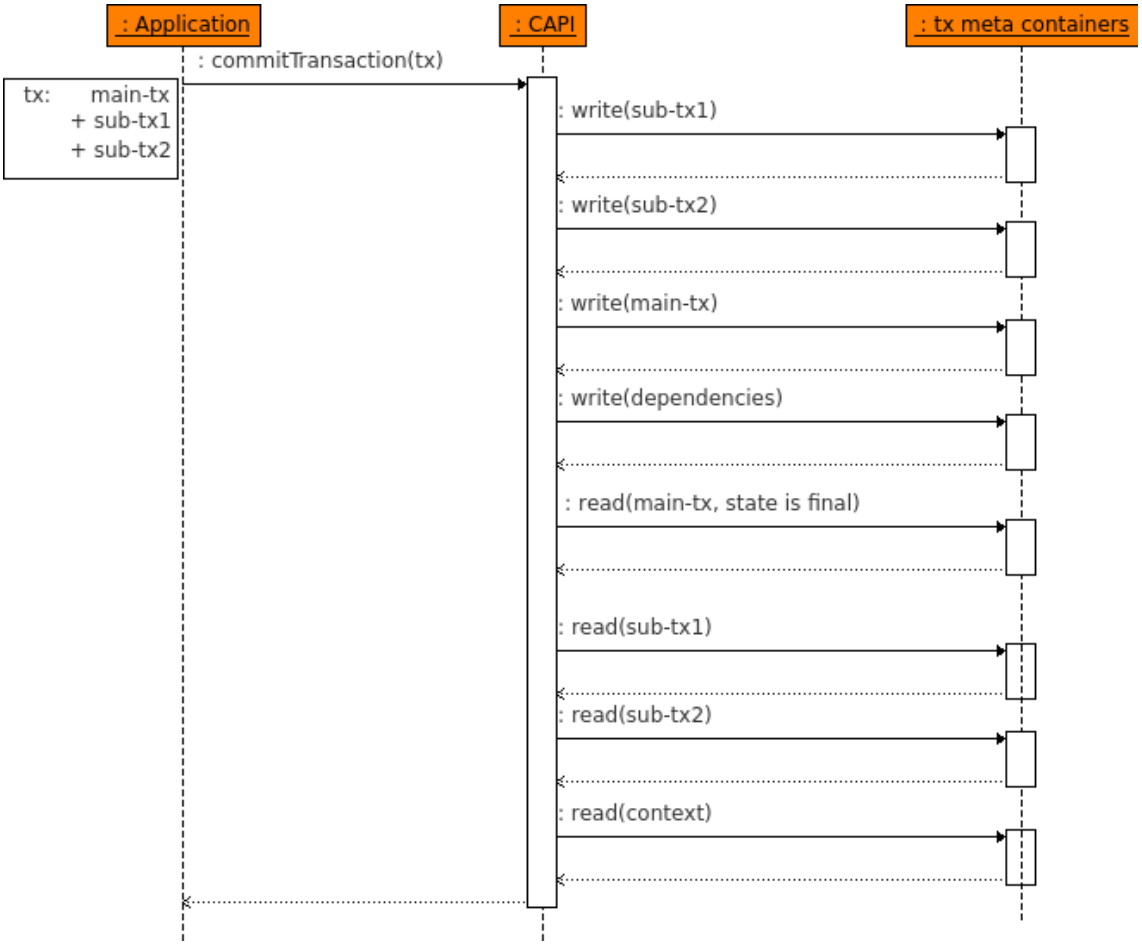

Figure 5.5: Sequence diagram of the `commitTransaction()`-method

5.2.2.1 Restrictions and Limitations

- Interfacing of 3rd party systems outside of the XVSM middleware is not explicitly supported
- All execution classes (`RexxExecutionCode`) of the transactions must be serializable
- All execution classes (`RexxExecutionCode`) must be available in the classpath of the execution space

5.2.2.2 The `AsyncTransactionManager`

With the `AsyncTransactionManager` the API is extended with the functionality of callbacks and therefore it has to be instantiated by the application. It provides two methods which are listed in table 5.14.

When the function `commitTransactionAsync()` is called by the application the `AsyncTransactionManager` triggers the commit of the transaction, creates a new thread and performs a blocking wait until the result is available. As soon as the transaction has finished it retrieves the result and calls the callback method `transactionCompleted` of the provided `RexxTransactionListener` class.

5.2.2.3 The monitoring API: `MAPI`

For the purpose of monitoring and debugging a separate API has been implemented - the `MAPI`. The difference to the standard `CAPI` is that the `MAPI` also returns intermediate results of running transactions, whereas the `CAPI` only returns the results for finished transactions.

The provided methods are listed in table 5.15.

5.2.3 The REXX transaction manager

The REXX transaction manager (`RexxTransactionManager`) acts as the operator of the REXX transaction processing system. It controls

- the startup and shutdown of transaction handlers,
- the startup and shutdown of guard processes and
- the registration of the aspect for the timeout handling (section 5.2.5).

5.2.3.1 Transaction handlers

The transaction handler (`RexxTransactionHandler`) is the core component of the implementation. It implements both workflows of the REXX model, the normal transaction workflow (figure 4.6) as well as the compensation workflow (figure 4.7). The transaction handler primarily operates on the `__rexx_data` container. It performs a blocking `take()` operation on the container and waits until a transaction is available for processing.

The operation starts when a new entry (`RexxDependencyEntry`) is available. Transaction handlers only operate on a single entry at the same time. One instance of the workflow is executed within one transaction. This assures that the system is always in a consistent state. There is only one exception to this approach. If transaction entries are ready for execution and have been set to state *running* or *running_compensation*, the

current transaction is committed. A new transaction is created and the timeout of the transaction is set according to the `timeout` attribute of the entry (section 5.2.5). This transaction is then passed to the execution code of the entry. After the execution has successfully finished, the same transaction is used to update the parent transaction and all successor transactions.

Figure 5.6 shows the coarse sequence diagram of the transaction handler. In case of leaf transactions the workflow of the transaction handler is internally split into two parts with one local transaction each. The first local transaction sets the state of the entry to *running*. This is required for monitoring applications to distinguish between *waiting for execution* and *running* transactions and is useful especially for long lived transactions.

The scope of the second transaction spans over the transaction's execution as well as over the updates of dependent transactions and the update of the context.

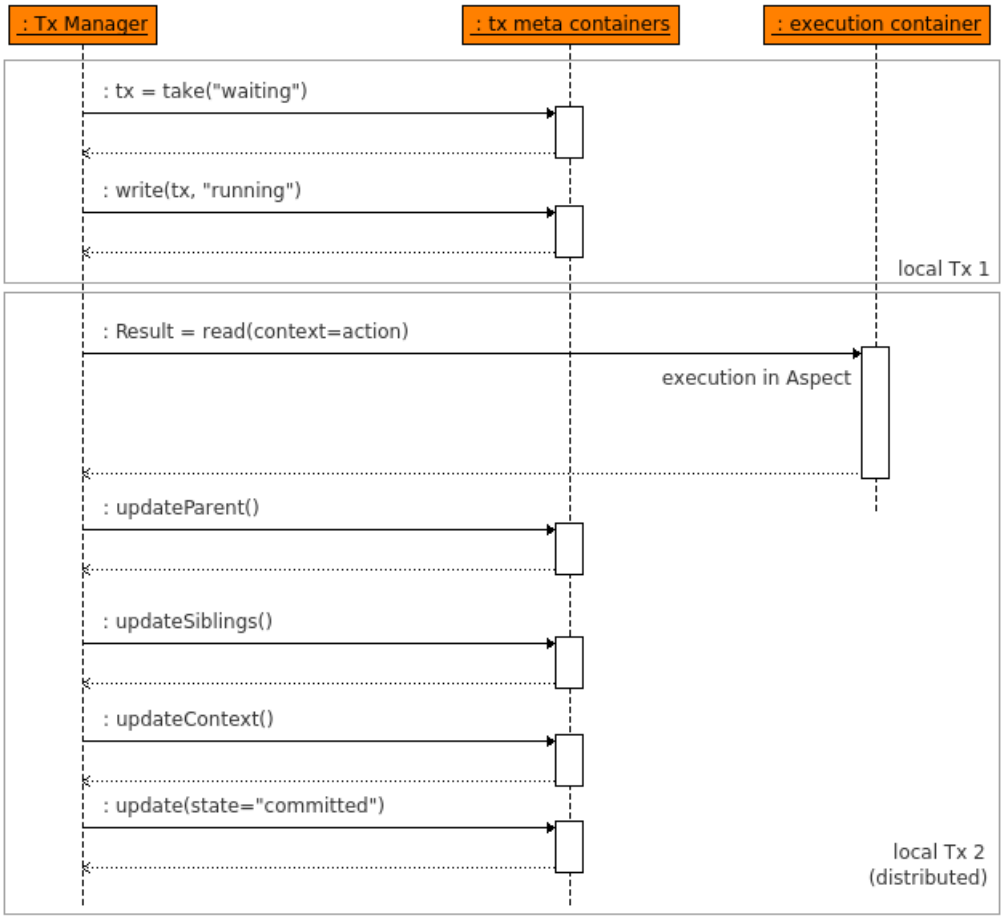

Figure 5.6: Sequence diagram of the transaction manager

This approach has the advantage that the timeout of the second local transaction can be set according to the specified *timeout* attribute. This would not be possible with only a single transaction because the timeout for the execution of the `RexxDependencyEntry` is not available when the transaction is created. The first local transaction is created before the blocking `take()` call.

However the approach results in an inconsistent state when a transaction manager crashes during the execution of an entry. In that case the entry remains in state *running* or *running compensation* and would never be processed by any transaction handler. This is the point where *Guard Processes* come into play.

5.2.3.2 Remote Execution

The REXX model supports the execution of leaf transactions on a remote space. The execution is performed by a dummy `read()` operation on a special container on the remote space. Therefore the `RexxExecutionAspect` is plugged to the interception point (`IPoint`) before and after the `read()` operation. The hook before the operation returns `SKIP` to prevent the execution of the real `read()` operation. The hook after the operation executes the passed leaf transaction. For the execution the REXX manager has to pass the following required information through the `RequestContext` of the operation:

- the available `RexxContext`
- the object `RexxExecutionCode` which contains the code
- the `TransactionReference` which has to be used
- the flag `compensate` which specifies the method which has to be executed, either the normal transaction or the compensation.

The dummy `read()` operation itself is executed using the local core without a transaction.

5.2.3.3 Guard processes

The only task of guard processes is to detect orphan transactions. These are transactions in state *running* or *running compensation* which are not handled by any transaction handler. Guard processes are implemented in the class `RexxTransactionGuard` and use the following pattern to filter the desired entries:

- The state is either *running* or *running compensation*.
- The `deadline` is earlier than the current time plus a certain offset.

The offset has been introduced to prevent unnecessary rollbacks of almost completed transactions. When a transaction entry matches the specified filter above, then the guard process resets the state to *waiting for execution* respectively *waiting for compensation* so that it will be re-evaluated by the next free transaction handler. Since the timeout is already elapsed, the transaction handler either aborts the transaction (if the state is *waiting for execution*) or it will set it to *compensation failed* (otherwise).

5.2.3.4 The `RexxAggregateFunction`

Parent transactions are using a `RexxAggregateFunction` to determine their state according to the state of the sub transactions. When all sub transactions are in a final state, method `apply(Map<String,RexxState> resChildren)` is called. The argument is a map with the results of the sub transactions indexed with their names (attribute `subid`). The result is either `True` or `False`. Currently there are two different aggregate functions supported:

- **AND** - The result is `True` if all sub transactions have committed. This is the default function if the aggregate function has not been set explicitly.
- **OR** - The result is `True` if at least one sub transaction has committed.

Depending on the result the next action in the workflow is processed (figure 4.6).

5.2.3.5 Remote Execution

The logic of a REXX transaction is coded in a Java class which is then handed over to the transaction handler. Since the transaction handler can be located on a remote machine it has to be ensured that all required classes of type `RexxExecutionCode` are within the classpath of the remote environment.

5.2.3.6 Configuration

The configuration of the REXX transaction manager is done in class `RexxConfiguration`, which is part of the MozartSpaces configuration `CommonsXmlConfiguration`. The configuration can be specified either directly in the application or through the XML configuration file. The configuration contains the following values:

- **Number of Handlers** - This value specifies the number of parallel running transaction handlers (`RexxTransactionHandler`). The default value is 5.
- **Number of Guards** - This value specifies the number of parallel running guard processes (`RexxTransactionGuard`). The default value is 1.

Listing 5.2 shows a sample XML configuration for the REXX transaction manager.

```
<mozartspacesCoreConfig>
    <!-- ... normal MozartSpaces config is here ... -->
    <rexx>
        <nbrHandlers>3</nbrHandlers>
        <nbrGuards>1</nbrGuards>
    </rexx>
</mozartspacesCoreConfig>
```

Listing 5.2: Sample REXX configuration

5.2.4 Concurrency and Scalability

There is neither direct communication among transaction handlers nor among guard processes. The synchronization and coordination is accomplished with the special containers (table 5.11) of the space. Therefore it is possible to run several instances of `RexxTransactionHandler` and `RexxTransactionGuard` without additional coordination effort. In theory this approach scales linearly with the number of handlers. But in practice the following factors limit the scalability:

- synchronized blocks within MozartSpaces
- number of threads/cores provided by the hardware
- internal structure of transactions (dependencies between transactions limit the performance)

5.2.5 Timeout Handling

The timeout of a transaction can be specified by the application with the attribute `timeout` in the object `RexxTransaction`. The timeout can be set for each transaction and sub transaction individually. Additionally the application can define a global timeout in the `commitTransaction()` operation of the `CAPI`. The value overrules all individual timeouts in case they are in conflict.

Every single `RexxDependencyEntry` has two attributes which control the internal timeouts:

- `deadline` specifies the absolute point in time when the transaction becomes invalid. This value is first set when the entry is written into the `__rexx_data` container and is updated before the execution is started.
- `timeout` is set by the application and will never be modified.

Whenever absolute time values are used, it is required that the clocks of all involved systems are synchronized. Since this is a very hard assumption, the approach was to reduce the number of involved systems to one - the space of the transaction manager. This has been accomplished with the container aspect `RexxContainerAspect` which has been added to container `__rexx_data` at the `IPoint` `postWrite`. The aspect only processes entries which have been directly written by the `commitTransaction()` operation of the `CAPI`. This is accomplished with a certain property which is passed through the `RequestContext` of the `write()` operation. This aspect initializes `deadline` and sets it to the current system time plus the global timeout.

When the transaction sets a transaction entry into state *running*, it also updates the field `deadline` according to formula 4.3. Afterwards it commits the current transaction and creates a new one. The timeout of the new transaction is then set to `deadline` minus the current system time. Afterwards the transaction is passed to the execution code. If the transaction times out, a `MzsTimeoutException` is thrown and the transaction handler sets the transaction entry to state *aborted*.

The field `deadline` is also periodically checked by guard processes (section 5.2.3.3).

5.2.5.1 Compensation timeout

For every transaction and sub transaction an individual compensation timeout can be specified. This timeout is independent of the normal timeout and is not affected by the global timeout which was passed by the `commitTransaction()` operation. When the transaction handler sets a transaction entry to state *running compensation* it also updates the field `deadline` to the current system time plus the specified compensation timeout and commits the current transaction. Then it creates a new transaction with the compensation timeout and passes it to the execution of the compensation code.

Since the global timeout is not considered a compensated transaction can take longer than the specified global timeout.

5.2.5.2 Deadlock Prevention

In the REXX model there are two scenarios where a deadlock between transaction handlers can occur.

In the first scenario we assume a transaction AB which consists of two sub transactions A and B where B depends on A. Initially all transactions are in state *waiting for execution*. Then the following schedule is executed by two transaction handler ($TH1$ and $TH2$):

1. $TM1$ performs a `take` on transaction AB
2. $TM2$ performs a `take` on transaction A and executes it

3. $TM2$ wants to update its successor B and its parent AB and therefore performs a `take`. The operation blocks because AB is locked by $TM1$

4. $TM1$ wants to check the state of the sub transactions and performs a `read` on transactions A and B. The operation blocks because A is locked by $TM2$

This conflict has been resolved in the method where the parent retrieves the status of the sub transactions. The request timeout of the `read` operation has been set to `RequestTimeout.ZERO`. This causes a `EntryLockedException` to be thrown when the desired entry is locked. This exception is caught and the state of the parent is reset to *suspended*. This does not affect the correctness of the model because in this case there is always at least one sub transaction of the parent which has not yet been finished. This sub transaction will then wake-up the parent again.

For the second scenario we assume the same transaction as in the first scenario, but now assume the following execution schedule:

1. $TM1$ performs a `take` on transaction A and executes it

2. $TM2$ performs a `take` on transaction B and checks its predecessors

3. $TM2$ performs a `take` on transaction A. The operation blocks because A is locked by $TM1$

4. $TM1$ wants to update its successor B and its parent AB and therefore performs a `take`. The operation blocks because B is locked by $TM2$

This conflict has been resolved in the method where the state of the predecessors is retrieved. Again the request timeout of the `read` operation has been set to `RequestTimeout.ZERO` and therefore the state is reset in case a predecessor is already locked. This does not affect the correctness of the model because in this case there is always at least one predecessor which has not yet been finished. And this predecessor will then wake up the transaction again.

Attribute	Description
`String appid`	label which specifies the owner or application of the transaction. The `appid` can be used to retrieve all open transactions of a certain application.
`String subid`	label which identifies the transaction within the application. It is used to retrieve the result of a certain parent or sub transaction.
`ArrayList <RexxTransaction> lstSubTx`	specifies all sub transactions.
`RexxTransaction parent`	references the parent transaction.
`Long timeout`	specifies the timeout in milliseconds for the transaction. See section 5.2.5 for further information regarding timeout handling.
`Long compensationTimeout`	specifies the timeout in milliseconds for the compensation of the transaction. See section 5.2.5 for further information regarding timeout handling.
`RexxExecutionCode execution`	class which contains the code for the execution of the transaction and the compensation. This argument is only considered for leaf transactions.
`ArrayList<RexxTransaction> lstDeps`	list of predecessor transactions.
`RexxAggregateFunction fntAggregate`	class which defines how the state is determined based on the states of the sub transactions. This argument is only considered for parent transactions.
`boolean forceExecution`	specifies whether the transaction is executed although not all sub transactions have been executed successfully. This argument is only considered for leaf transactions.

Table 5.10: The `RexxTransaction` class

Container name	Persistent	Description
`__rexx_meta`	Yes	stores the next transaction ID
`__rexx_data`	Yes	holds an entry for every single sub transaction. This is the central operational container of the transaction handlers.
`__rexx_context`	Yes	stores all data which has been written by transactions and compensations.
`__rexx_dependencies`	Yes	stores the relations between sub transactions.

Table 5.11: List of containers used in the REXX Implementation

Method	Description
`commitTransactionAsync (rexxTx, space)`	passes `transaction` to the transaction handler and returns immediately the transaction reference. Argument `space` specifies the space of the transaction handler
`waitForTransaction( refTx, timeout)`	waits until transaction with reference `refTx` has finished and returns the result. Argument `timeout` specifies the timeout for this blocking call
`commitTransaction( rexxTx, space)`	calls the methods `commitTransaction-Async` and `waitForTransaction` sequentially
`getOpenTransactions( appid, space)`	returns the list of transactions whose application ID matches the parameter `appid` and whose result has not been queried so far.

Table 5.12: Additional methods in the `CAPI`

Attribute	Description
`String appid`	label which specifies the owner or application of the transaction
`String subid`	label which identifies the transaction within the application
`ArrayList <RexxResult> lstSubTx`	contains the results of all sub transactions
`RexxResult parent`	references the result of the parent transaction
`Map<String, Object> context`	contains all context variables which have been written by the sub transactions
`RexxState state`	any of the four final states. The enumeration `RexxState` contains all states which are shown in figure 4.4
`String id`	internal ID of the transaction, which is never used by the application

Table 5.13: The attributes of the `RexxResult` class

Method	Description
`commitTransactionAsync (transaction, space, listener)`	has the same functionality as the function in the `CAPI`. The argument `listener` specifies a `RexxTransactionListener` class.
`shutdown()`	stops all running notification threads

Table 5.14: Provided methods of the `AsyncTransactionManager`

Method	Description
`getAllTransactions()`	returns the current internal status of all transactions of the transaction manager
`getAllTransactions (appid)`	has the same functionality as `getAllTransactions()` but only returns transactions of a certain application
`getRunningTransactions ()`	returns the current internal status of all running transactions of the transaction manager
`getRunningTransactions (appid)`	has the same functionality as `getRunningTransactions()` but only returns transactions of a certain application

Table 5.15: Provided methods of the `MAPI`

CHAPTER 6

Evaluation

6.1 Evaluation of usability

Microsoft defines usability as ".. a measure of how easy it is to use a product to perform prescribed tasks." [6] First we have to distinguish between usability and utility. Utility refers to the amount of tasks which can be performed with a certain system. The more tasks it can perform the higher is the utility. Regarding computation the utility of programming languages is very high since nearly every task can be performed. However a simple calculator has a low utility since it is restricted to very limited functionality. Utility does not consider the effort which is required to perform a certain task.

Usability on the other hand traditionally refers to the following attributes [6]:

- *Discovery* is related to the time how long it takes for a new user to discover a certain functionality. In this thesis this new functionality would be the support of distributed transactions.
- *Learning* starts when a new functionality has been discovered. It focuses on the amount of time a user requires to perform a certain task after he or she has discovered the required functionality.
- *Efficiency* comes into play when a user has full knowledge of a certain functionality. It defines the effort which is required by a trained user to perform a certain task.

In this section we focus on the usability of distributed transactions of MozartSpaces. The evaluation of the attributes *Discovery* and *Learning* would require a detailed study with application programmers and that is beyond the scope of this thesis. However we

want to focus on the efficiency of the implementation of the newly developed transaction models. Therefore we compare the number of lines of code which is required to perform a certain functionality. Additionally we address the quality of the produced code of the new transaction models. The quality of the code directly impacts the maintainability of an application. For the evaluation of the efficiency and maintainability we assume the following requirement:

Data shall be replicated to three different locations. Related to MozartSpaces this means to write entries to three different remote spaces on the network.

First a reference implementation is shown which implements this functionality with the use of local transactions of the current implementation of MozartSpaces. Afterwards the same functionality is implemented with REXX transactions as well as with the Extended Paxos Commit protocol.

6.1.1 Reference Implementation

The reference implementation uses three different local transactions to perform the writes to the three remote space. Listing 6.1 shows the code of this implementation where $s1$, $s2$, $s3$ are the space identifiers, $strC$ is the name of the remote container and $capi$ is an instance of the CAPI of MozartSpaces.

The method `replicationalWrite` performs a lookup and afterwards writes the entry into the remote container. Method `compensateWrite` is responsible to compensate the written entry and therefore deletes the entry from the remote container. Both methods execute their operation in the context of a local transaction. The main method `referenceReplication` coordinates the workflow of the replication. It calls `replicationalWrite` for each space and has to trigger the compensation actions if any of the methods fails. If the entry has already been written to space $s1$ and the write to space $s2$ fails, then the entry in space $s1$ has to be deleted by calling `compensateWrite`. Otherwise the system would become inconsistent.

```
1 public void referenceReplication() throws MzsCoreException {
2     boolean cont = true;
3     try {
4         replicationalWrite(s1,  strC);
5     } catch (MzsCoreException e) {
6         cont = false;
7     }
8     if (cont) {
9         try {
```

```
10              replicationalWrite(s2,  strC);
11          } catch (MzsCoreException e) {
12              compensateWrite(s1, strC);
13              cont = false;
14          }
15          if (cont) {
16              try {
17                  replicationalWrite(s3,  strC);
18              } catch (MzsCoreException e) {
19                  compensateWrite(s1, strC);
20                  compensateWrite(s2, strC);
21                  cont = false;
22              }
23          }
24      }
25  }
26
27  public void replicationalWrite(URI space, String cName)
28          throws MzsCoreException {
29      TransactionReference tx = capi.createTransaction(3000, space);
30      ContainerReference c = capi.lookupContainer(strC, space, 3000, tx
            );
31      capi.write(new Entry("TestEntry"), c, 3000, tx);
32      capi.commitTransaction(tx);
33  }
34
35  public void compensateWrite(URI space, String cName)
36          throws MzsCoreException {
37      TransactionReference tx = capi.createTransaction(3000, space);
38      ContainerReference c = capi.lookupContainer(strC, space, 3000, tx
            );
39      Selector sel = AnyCoordinator.newSelector(1);
40      capi.delete(c, Arrays.asList(sel), RequestTimeout.INFINITE, tx);
41      capi.commitTransaction(tx);
42  }
```

Listing 6.1: Implementation of replication using the local transaction system

Of course, this implementation does not preserve consistency if the system crashes between two commit calls. However, since it at least provides the shown complexity we can use it as reference for the comparison of the usability. An implementation which also covers correct recovery actions would be much more complex.

6.1.2 Implementation with Extended Paxos Commit

The implementation of the replication functionality with the Extended Paxos Commit protocol is much simpler because due to the fact that it is an atomic commit protocol,

it does not require compensation. Furthermore in contrast to the reference implementation, only a single transaction is required for the whole replication. Due to these two facts the number of lines of code for the implementation with Extended Paxos Commit is very low (see listing 6.2).

```
1  public void paxosReplication() throws MzsCoreException {
2      PaxosTransactionReference tx = capi.createTransaction(3000, s1);
3      paxosWrite(s1, strC, tx);
4      paxosWrite(s2, strC, tx);
5      paxosWrite(s3, strC, tx);
6      capi.commitTransaction(tx);
7  }
8
9  public void paxosWrite(URI space, String cName,
       PaxosTransactionReference tx) throws MzsCoreException {
10     ContainerReference c = capi.lookupContainer(strC, space, 3000, tx
           );
11     capi.write(new Entry("TestEntry"), c, 3000, tx);
12 }
```

Listing 6.2: Implementation of replication using Extended Paxos Commit

6.1.3 Implementation with REXX transactions

This section shows how the same functionality could be implemented with the use of REXX transactions. In the first step a new class for the execution code has to be created (see listing 6.4). This class defines the operations of the transaction as well as its compensation. In method `rexxReplication` of listing 6.3 a new `RexxTransaction` is created. The transaction contains three sub transactions, one for each remote space. Afterwards the whole transaction is committed and the result is put into variable res. If the commit has not been successful, the class `RexxResult` provides the following useful methods to detect the erroneous sub transactions:

- `getAbortedTransactions` returns all aborted leaf transactions
- `getCompensatedTransactions` returns all compensated parent and sub transactions
- `getFailedTransactions` returns all parent and sub transactions which caused state *failed*

The result also contains the corresponding error messages of all *aborted* or *failed* transactions.

```
1  public void rexxReplication() throws MzsCoreException {
2      RexxTransaction main = new RexxTransaction(3000L);
3      main.addSubTransaction(new RexxTransaction(
4              new ReplicationalWrite(s1, strC)));
5      main.addSubTransaction(new RexxTransaction(
6              new ReplicationalWrite(s2, strC)));
7      main.addSubTransaction(new RexxTransaction(
8              new ReplicationalWrite(s3, strC)));
9      RexxResult res = capi.commitTransaction(main);
10 }
```

Listing 6.3: Implementation of replication using REXX transactions

```
1  public class ReplicationalWrite extends RexxExecutionCode {
2      private static final long serialVersionUID = 1L;
3      private URI space;
4      private String container;
5
6      public ReplicationalWrite(URI space, String container) {
7          this.space = space;
8          this.container = container;
9      }
10
11     @Override
12     public boolean execute(Capi capi, TransactionReference tx,
           RexxContext context)
13             throws MzsCoreException {
14         ContainerReference c = capi.lookupContainer(container, space,
               RequestTimeout.INFINITE, tx);
15         capi.write(new Entry("TestEntry"), c, RequestTimeout.INFINITE
               , tx);
16         return true;
17     }
18
19     @Override
20     public boolean compensate(Capi capi, TransactionReference tx,
21             RexxContext context) throws MzsCoreException {
22         ContainerReference c = capi.lookupContainer(container, space,
               RequestTimeout.INFINITE, tx);
23         Selector sel = AnyCoordinator.newSelector(1);
24         capi.delete(c, Arrays.asList(sel), RequestTimeout.INFINITE,
               tx);
25         return true;
26     }
27 }
```

Listing 6.4: Definition of class `ReplicationalWrite`

6.1.4 Comparison of the implementations

Efficiency The reference implementation has about 40 lines of code. The effort of the implementation with REXX transactions is about the same or slightly lower, but it would be much lower if the error handling would have been included into the implementations. The reference implementation would have to catch and handle all possible exceptions whereas in the REXX transaction model all exceptions are handled by the transaction manager. Such exceptions could be timeout exceptions or any other kind of core exceptions (`MzsCoreException`) which are thrown during the execution of transactions.

All errors of a REXX transaction can be extracted from the `RexxResult` class which is returned by the commit method.
The implementation with Extended Paxos Commit is the most effective implementation. It only requires about 10 lines of code to perform the same functionality.

Maintainability The reference implementation is the most complex one since its main method coordinates the individual transactions as well as their compensation. In contrast the main method of the REXX implementation is very simple and the code of the transaction and the compensation are defined in a separate class. This improves maintainability because all related code is located at the same place.
The implementation with Extended Paxos Commit is also very simple because no compensation is required.

6.2 Performance evaluation

We now focus on the performance of the distributed transaction models. Therefore we create a certain scenario and evaluate the execution times of the different implementations. Figure 6.1 shows the setup which is used for the performance benchmarks. The hardware consists of the following parts:

- The PC on *Site 1* has the following characteristics:
  - Ubuntu 12.04
  - Intel Core i5-450M CPU (2 Cores, 4 Threads, 2.4GHz)
  - 4GB RAM
  - Intel SATA solid-state drive (SSD) SSDSA2M160 (160GB)

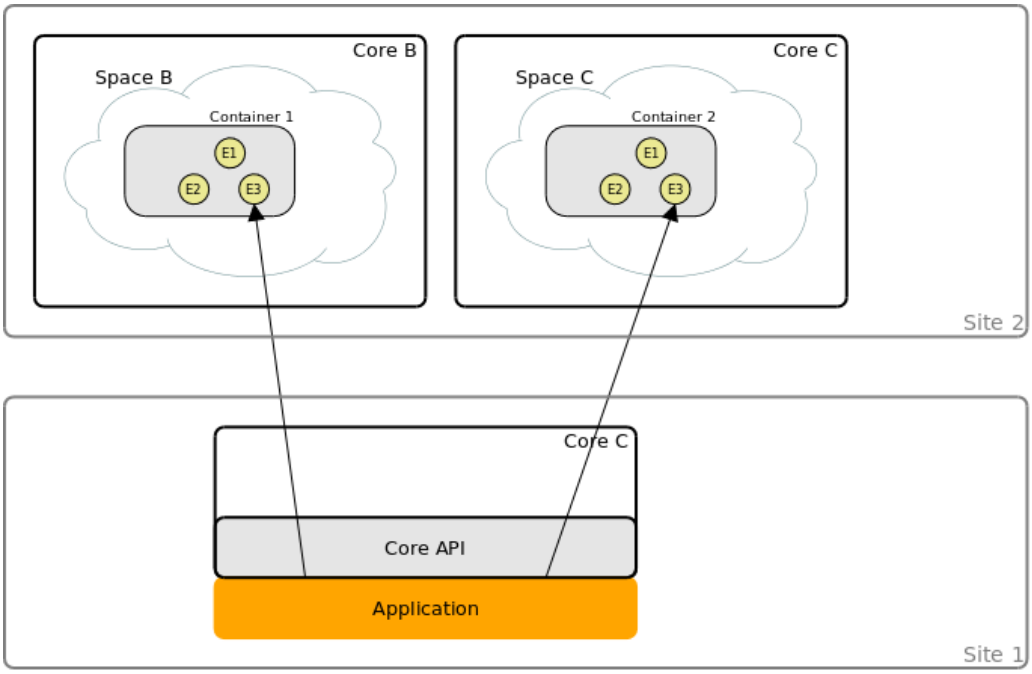

Figure 6.1: Scenario of the benchmark

- The PC on *Site 2* has the following characteristics:
  - Windows 7 Home Premium
  - Intel Core i3-2370M CPU (2 Cores, 4 Threads, 2.4GHz)
  - 4GB RAM
  - Seagate SATA harddisk ST9750423AS (750GB, 5400 RPM, 16MB Cache)
- Both hosts (*Site 1* and *Site 2*) are connected through a WLAN network.

The scenario is a simple replication where distributed transactions are required. The application is located on *Site 1* and does not use an embedded space. *Site 2* hosts two spaces each consisting only of one container which is coordinated with an *AnyCoordinator*. One cycle of the benchmark consists of four operations:

1. lookup of *Container 1* on *Space B*,
2. write of one entry into *Container 1*,
3. lookup of *Container 2* on *Space C* and
4. write of one entry into *Container 2*.

The benchmark scenario is performed with the same three transaction models which have already been compared in terms of usability. Every execution calls the mentioned cycle 50 times and is performed with the two persistency profiles (see section 3.2.2) *Lazy* and *Transactional with fsync*. The persistency profile is always the same for all 3 cores.

Local transactions - This is the implementation which uses the current transaction functionality of MozartSpaces. Since it does not support distributed transactions, consistency is not guaranteed in this case. However we will use this implementation as reference for the comparison with the other two transaction models. This implementation uses one local transaction for each remote space. So the first two operations are executed with a local transaction of *Space B* and the second two operations are performed using a local transaction of *Space C*. As an additional reference case the benchmark has also been performed without transactions.

Paxos Commit - Since this transaction model supports distributed transactions, all 4 operations are performed with a single transaction. The execution of this benchmark uses two acceptor instances which are located on *Core A*. The benchmark of Paxos Commit has been executed with the following two approaches:

- **Synchronous approach** - The commit is performed with the blocking `commitTransaction` method. Therefore all 50 cycles are executed sequentially. After the commit operation of the first cycle has been completed, the second cycle is started.
- **Asynchronous approach** - The commit is performed with the non-blocking `commitTransactionAsync` method. For the indication that all commit operations have been completed successfully a `test()` operation has been used. This operation blocks until all 50 entries are available on the remote space.

REXX transaction model - This implementation uses a REXX transaction which consists of two sub transactions. One sub transaction processes the operations of *Space B* and the second sub transaction processes the operations of *Space C*. Due to the separation into two sub transactions both are executed in parallel by the transaction manager. Both sub transactions internally use the Extended Paxos Commit protocol with a single acceptor node. Distributed transactions are required for the execution of remote operations because additionally to the remote data also the internal data of the REXX transaction manager have to be modified with one transaction. The benchmark of the REXX model has been executed with the following two approaches:

- **Synchronous approach** - The commit is performed with the blocking `commitTransaction` method. Therefore all 50 cycles are executed sequentially. After

the commit operation of the first cycle has been completed, the second cycle is started.

- **Asynchronous approach** - The commit is performed with the non-blocking `commitTransactionAsync` method. After all commit operations have been executed, the results of all transactions are collected synchronously.

Both benchmarks have been evaluated with different numbers of transaction handlers. This should show at least one aspect of the scalability of the implementation.

6.2.1 Evaluation of the results

Figure 6.2 summarizes the results of the executions. The reference implementation using local transactions requires about 44s to complete the given scenario with the *Lazy* profile. In contrast if no transactions are used the execution time is only one third. This is because the local commit operation is very expensive.

The time required by local transactions can be reduced by about 35% if the synchronous approach of Paxos Commit is used. In contrast to the reference implementation only one instead of two transactions are required. The reason for this tremendous improvement is that the time which is required for the commit of local transactions is much higher than all other involved operations. In this case Paxos Commit has two advantages compared to the reference implementation:

1. The local commit operations are called asynchronously, which allows the application to continue before the local commit has finished.

2. The local transactions are committed in parallel whereas in the reference implementation they are committed sequencially.

The asynchronous approach of Paxos Commit has given the best result with an execution time of about 4s. This is about one third of the time which has been required by the execution without transactions. The reason for this big difference is that MozartSpaces implicitly creates a local transaction for every operation which does not have an explicit transaction. Thus the execution without transactions internally uses four local transactions for one cycle with four operations whereas the Paxos execution only uses one local transaction.

The benchmark of the REXX model with only one handler has given the worst result of all executions. This is because the REXX transactions internally use two Paxos Commit transactions and both transactions have to be executed sequentially because only one handler is available. The execution time of the synchronous approach can be reduced by about 25% if two handlers are used. However the result can not be further improved by adding additional handlers, since the REXX transaction in the benchmark scenario only consists of two sub transactions.

On the other hand figure 6.2 shows that the execution time of the asynchronous approach can be reduced by using additional transaction handlers. For example the execution with three handlers is about 3-4 times faster than the execution with one handler. Although the used hardware only supports four threads in parallel, there is still a difference between five handlers and eight handlers. This is because the work of the transaction handlers is I/O intensive and therefore the operating system reschedules other threads while the thread of the transaction handler is waiting for new data.

We now focus on the two persistency profiles which have been evaluated. The difference between *Lazy* and *Transactional with fsync* is not that high as stated in [Zar12]. The difference between the two profiles is less than 10% in all executed benchmarks. This because the hardware of *Site 1* is equipped with a solid-state drive.

The scenario which has been used for the evaluation of the performance is very simple. However in case of distributed transactions the performance of the new transaction models is much better than the performance of local transactions. The higher the complexity of a distributed transaction the greater is the different regarding the performance. An example of a more complex distributed transaction would be the already discussed booking of a trip (section 4.3).

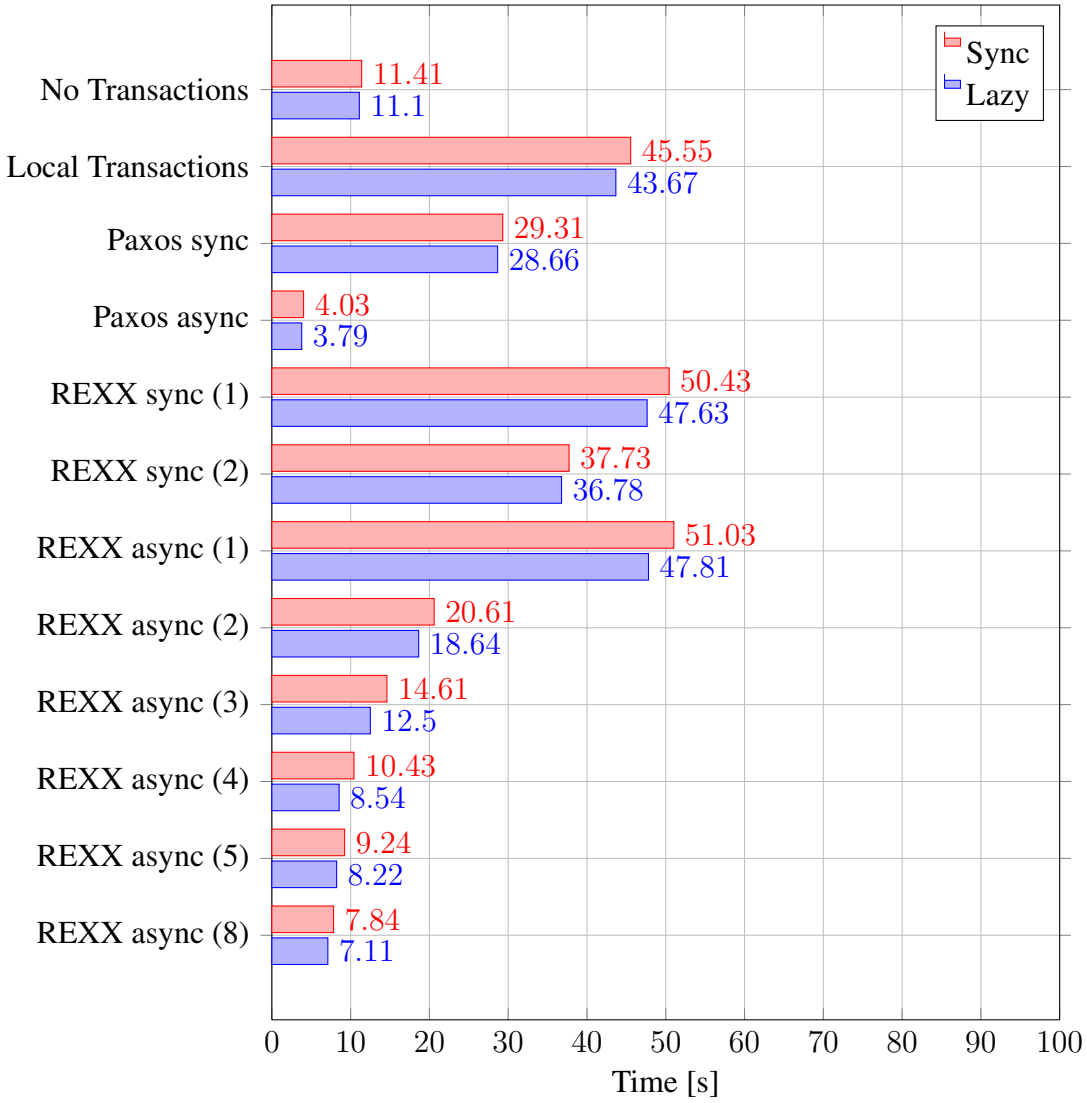

Figure 6.2: Processing time of the different implementations for the given replication scenario. The number in the parantheses specifies the number of parallel running transaction handlers.

CHAPTER 7

Future work

Both presented transaction models do not pro-actively detect and handle distributed deadlocks. In both models deadlocks are implicitly handled with timeouts. A new deadlock detection service could be implemented to pro-actively detect and resolve deadlocks. There are several algorithms provided in the literature [GdMFnG+99] [BH03].

For the Extended Paxos Commit algorithm there is some space for improvements in the following areas:

- **Leader Election** - The implemented leader election is based on the basic algorithm presented in [ADGFT01]. The drawback of this algorithm is its bad behavior in terms of stability. If only one of the used message links is faulty for more than the specified timeout period, a new leader is elected. In an unreliable network this behavior would cause many unnecessary changes of the leader and also decreases the liveness of the Paxos Commit algorithm. Therefore the same paper describes additional, more complex algorithms which lead to a better stability. Another enhanced leader election algorithm which is dedicated to Paxos is presented in [MOZ05].

- **Persistence of the Isolation Manager** - Since the data of the isolation manager is not included in the persistency a workaround with a space aspect has been implemented. Once the persistency covers the isolation manager the Paxos implementation will be cleaner. Therefore the internal state including entry locks, container locks and log items of transactions has to be written to the persistency backend as soon as the transations have been prepared.

- **Increase Performance** - There are several possibilities to improve the performance of the standard Paxos Commit algorithm. For example acceptor nodes

could directly send Phase3 messages to the resource managers and would so save one message delay. Other approaches are discussed in the literature [Lam06] [Moi11].

- **Synchronization of Paxos Commit** - The current implementation does not guarantee that all resource managers already have committed when the `commitTransaction` operation returns to the application. There has not yet been found a good solution to solve this issue.

Regarding REXX transactions there are also areas of improvements:

- **Monitor Application** - A graphical monitor application could be implemented to ease the debugging of REXX transactions. This application could use the provided `MAPI` to retrieve the internal state of REXX transactions.
- **Extended Dependencies** - The current implemetation does not directly support the `XOR` connection of sub transactions. By extending the functionality of dependencies the REXX transaction model could directly handle such connections.

CHAPTER 8

Conclusion

The goal of this work was to design a relaxed distributed transaction model for the XVSM middleware. The model should have both characteristics, high concurrency and a high degree of consistency. For this purpose the focus of this thesis has been put on two transaction models where each of them addresses one of the desired characteristics.

Extended Paxos Commit provides strict ACID compliance and therefore guarantees high consistency. The original protocol has been extended by sub transactions which can be combined by commit rules. Especially the sophisticated logical commit rule allows additional functionalities like function replication. In certain scenarios, like the operation on remote spaces, the MozartSpaces implementation of Paxos Commit outperforms the existing local transactions.

The REXX transaction model is aligned to the Flex model and to business processes which are defined by WS-BPEL (Web Services Business Process Execution Language). REXX transactions can be used to model complete business processes which consist of an arbitrary combination of activities (sub transactions). The execution is performed in the control of the REXX transaction manager which leads to several advantages. The application developer need not care about the coordination among sub transactions which improves the usability of the transaction model. Additionally sub transactions can be executed in parallel which leads to higher performance. In contrast to Paxos Commit the REXX transaction model provides a higher concurrency since sub transactions are committed immediately.

Both introduced transaction models can be combined to provide both characteristics, high concurrency and consistency, in a single transaction model. Therefore Extended Paxos Commit is used for the execution of sub transactions in the REXX transaction model so that ACID compliant distributed transactions can be executed. However it is not possible to have both desired characteristics in the same transaction. The performance of REXX transactions depends on the number of running transaction managers.

The evaluation shows that the performance of the REXX model is significantly higher than the synchronous execution of Paxos Commit if a sufficient number of transaction managers is used. When two REXX transaction managers are used the execution with the REXX model is 20% slower compared to Paxos, however if eight transaction managers are used the REXX model is about 70% faster. The execution with eight transaction managers is even 35% faster than the reference implementation which does not use explicit transactions.

With the design and the implementation of the new transaction models the XVSM middleware and especially MozartSpaces now support distributed transactions with several features. This allows the implementation of more complex business applications.

APPENDIX A

The REXX Transaction Model

MODULE $REXXtransaction$

EXTENDS $Integers$, $FiniteSets$

CONSTANTS $txPool$, Set of all transactions
$predecessors$, map predecessors $\{tx \rightarrow \{tx1, tx2\}\}$
$parents$ map of parents $\{tx \rightarrow Ptx, tx2 \rightarrow Ptx2\}$

VARIABLES $states$ The state of the transactions

Set of all possible states
$all\_states \triangleq$ {"waitingExec", "suspended", "running", "waitingComp", "runningComp", "committed", "aborted", "compensated", "compFailed"}

Set of final states
$final\_states \triangleq$ {"committed", "aborted", "compensated", "compFailed"}

The initial predicate
$RTInit \triangleq \quad \wedge \ (states = [t \in txPool \mapsto$ "waitingExec"$])$

Parent transaction
$parent(t) \triangleq parents[t]$

Set of all predecessors
$pred(t) \triangleq predecessors[t]$

Set of all successors
$succ(t) \triangleq \{tx \in txPool : t \in pred(tx)\}$

Set of predecessors which are in a final state

$final\_pred(t) \triangleq \{tx \in pred(t) : states[tx] \in final\_states\}$

All predecessors are in a final state

$pred\_ok(t) \triangleq Cardinality(pred(t)) = Cardinality(final\_pred(t))$

Checks whether there is a transaction with parent t

$isParent(t) \triangleq \exists\, tx \in txPool : (parent(tx) = t)$

Set of all sub transactions of t

$getChilds(t) \triangleq \{tx \in txPool : parent(tx) = t\}$

Set of all sub transactions of t which are in a final state

$getFinalChilds(t) \triangleq \{tx \in txPool : (parent(tx) = t \wedge states[tx] \in final\_states)\}$

Checks whether all sub transactions are in a final state

$childsFinished(t) \triangleq Cardinality(getChilds(t)) = Cardinality(getFinalChilds(t))$

Set current transaction to "s", Set parent transaction to "waitingExec", Set successor transactions to "waitingExec"

$update\_states(t,\, s) \triangleq [tx \in \text{DOMAIN } states \mapsto$
IF $tx \in succ(t) \vee tx = parent(t)$
THEN "waitingExec"
ELSE IF $tx = t$ THEN s ELSE $states[tx]]$

Set current transaction to "s", Set parent transaction to "waitingExec"

$update\_states\_comp(t,\, s) \triangleq [tx \in \text{DOMAIN } states \mapsto$
IF $tx = parent(t)$
THEN "waitingExec"
ELSE IF $tx = t$ THEN s ELSE $states[tx]]$

Set current transaction (parent) to "s", Set committed siblings to "waitingComp"

$compensate\_childs(t) \triangleq [tx \in \text{DOMAIN } states \mapsto$
IF $tx \in getChilds(t) \wedge states[tx] =$ "committed"
THEN "waitingComp"
ELSE IF $tx = t$ THEN "suspended" ELSE $states[tx]]$

We now define the actions that my be performed by leaf transactions

Set to running if all dependencies are met

$CRunning(t) \triangleq \wedge states[t] =$ "waitingExec"
$\wedge \neg isParent(t)$
$\wedge pred\_ok(t)$
$\wedge states' = [states \text{ EXCEPT } ![t] =$ "running"$]$

Set to suspended if not all dependencies are met

$$CSuspended(t) \triangleq \land states[t] = \text{“waitingExec”} \land \neg isParent(t) \land \neg pred\_ok(t) \land states' = [states \text{ EXCEPT } ![t] = \text{“suspended”}]$$

Wake up parent and all successors

$$CAbort(t) \triangleq \land states[t] = \text{“running”} \land states' = update\_states(t, \text{“aborted”})$$

Wake up parent and all successors

$$CCommit(t) \triangleq \land states[t] = \text{“running”} \land states' = update\_states(t, \text{“committed”})$$

We now define the actions that my be performed by parent transactions

Commit Parent transaction

$$PCommit(t) \triangleq \land states[t] = \text{“waitingExec”} \land isParent(t) \land childsFinished(t) \land \forall\, tx \in getChilds(t) : states[tx] = \text{“committed”} \land states' = update\_states(t, \text{“committed”})$$

Abort Parent transaction

$$PAbort(t) \triangleq \land states[t] = \text{“waitingExec”} \land isParent(t) \land childsFinished(t) \land \forall\, tx \in getChilds(t) : states[tx] = \text{“aborted”} \land states' = update\_states(t, \text{“aborted”})$$

$$PCompensated(t) \triangleq \land states[t] = \text{“waitingExec”} \land isParent(t) \land childsFinished(t) \land \forall\, tx \in getChilds(t) : states[tx] \in \{\text{“aborted”}, \text{“compensated”}\} \land \exists\, tx2 \in getChilds(t) : states[tx2] = \text{“compensated”} \land states' = update\_states\_comp(t, \text{“compensated”})$$

$$PFailed(t) \triangleq \land states[t] = \text{“waitingExec”} \land isParent(t) \land childsFinished(t) \land \exists\, tx \in getChilds(t) : states[tx] = \text{“compFailed”} \land states' = update\_states\_comp(t, \text{“compFailed”})$$

Compensate transaction

$PCompensate(t) \triangleq \land states[t] = \text{“waitingExec”}$
$\land isParent(t)$
$\land childsFinished(t)$
$\land \exists\, tx \in getChilds(t) : states[tx] = \text{“aborted”}$
$\land \exists\, tx \in getChilds(t) : states[tx] = \text{“committed”}$
$\land states' = compensate\_childs(t)$

$PSuspended(t) \triangleq \land states[t] = \text{“waitingExec”}$
$\land isParent(t)$
$\land \neg childsFinished(t)$
$\land states' = [states \text{ EXCEPT } ![t] = \text{“suspended”}]$

We now define the actions that my be performed by both:
leaf transactions and parent transactions

does not consider compensation dependencies

$RunningComp(t) \triangleq \land states[t] = \text{“waitingComp”}$
$\land states' = [states \text{ EXCEPT } ![t] = \text{“runningComp”}]$

$CompensationOk(t) \triangleq \land states[t] = \text{“runningComp”}$
$\land states' = update\_states\_comp(t, \text{“compensated”})$

$CompensationFailed(t) \triangleq \land states[t] = \text{“runningComp”}$
$\land states' = update\_states\_comp(t, \text{“compFailed”})$

Definition of the next actions

$RTNext \triangleq \lor \exists\, t \in txPool :$ The next-state action
Leaf transactions
$\lor CSuspended(t)$
$\lor CRunning(t)$
$\lor CCommit(t)$
$\lor CAbort(t)$
Parent transactions
$\lor PCommit(t)$
$\lor PAbort(t)$
$\lor PSuspended(t)$
$\lor PCompensate(t)$
$\lor PCompensated(t)$
$\lor PFailed(t)$
Both

$\vee\ RunningComp(t)$
$\vee\ CompensationOk(t)$
$\vee\ CompensationFailed(t)$

Definition of the invariants

Consistency: If parent is final, then also all sub transactions are final

$RTConsOK \triangleq \forall\, t \in txPool :$
$(isParent(t) \wedge states[t] \in final\_states) \Rightarrow childsFinished(t)$

Type Invariant

$RTTypeOK \triangleq \wedge\ states \in [txPool \rightarrow all\_states]$

If the parent is committed, then also all childs must be committed.
This model only considers the default AND aggregate function

$RTConsCommitted \triangleq \forall\, t \in txPool : (isParent(t) \wedge states[t] =$ "committed")
$\Rightarrow \forall\, p \in getChilds(t) : states[p] =$ "committed"

If the parent is in state 'aborted' then all sub transactions must
also be in state 'aborted'

$RTConsAborted \triangleq \forall\, t \in txPool : (isParent(t) \wedge states[t] =$ "aborted")
$\Rightarrow \forall\, p \in getChilds(t) : states[p] =$ "aborted"

If a transaction is in state 'failed', then the parent must also be in
state 'failed' if it is in a final state

$RTFailed \triangleq \forall\, t \in txPool : (states[t] =$ "compFailed"
$\wedge\ parent(t) \in txPool$
$\wedge\ states[parent(t)] \in final\_states)$
$\Rightarrow\ states[parent(t)] =$ "compFailed"

Specification

$vars \triangleq \langle states \rangle$
$RTSpec \triangleq\ RTInit \wedge \Box[RTNext]_{vars}$

Apply the invariants to the specification

THEOREM $RTSpec \Rightarrow\ \wedge \Box RTConsOK$
$\wedge \Box RTTypeOK$
$\wedge \Box RTConsCommitted$
$\wedge \Box RTConsAborted$
$\wedge \Box RTFailed$

Bibliography

[ADGFT01] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 108–122, London, UK, UK, 2001. Springer-Verlag.

[BACF08] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. Depspace: a byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 163–176, New York, NY, USA, 2008. ACM.

[Bar10] Martin-Stefan Barisits. Design and implementation of the next generation xvsm framework. Master's thesis, Institut für Computer Languages, Vienna University of Technology, 2010.

[BEeK93] Omran A. Bukhres, Ahmed K. Elmagarmid, and eva Kühn. Implementation of the flex transaction model. *IEEE Data Eng. Bull.*, pages 28–32, 1993.

[BH03] S. Bhalla and M. Hasegawa. Automatic detection of multi-level deadlocks in distributed transaction management systems. In *Parallel Processing Workshops, 2003. Proceedings. 2003 International Conference on*, pages 297–304, 2003.

[Bre10] Eric Brewer. A certain freedom: thoughts on the cap theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 335–335, New York, NY, USA, 2010. ACM.

[CK12] Stefan Craß and Eva Kühn. A coordination-based access control model for space-based computing. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1560–1562, New York, NY, USA, 2012. ACM.

[CPW07] Lásaro Camargos, Fernando Pedone, and Marcin Wieloch. Sprint: a middleware for high-performance transaction processing. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 385–398, New York, NY, USA, 2007. ACM.

[Cra10] Stefan Craß. A formal model of the extensible virtual shared memory (xvsm) and its implementation in haskell : design and specification. Master's thesis, Institut für Computer Languages, Vienna University of Technology, 2010.

[Dö11] Tobias Dönz. Design and implementation of the next generation xvsm framework: runtime, protocol and api. Master's thesis, Institut für Computer Languages, Vienna University of Technology, 2011.

[eK00] eva Kühn. Coordination system. *European Patent, PCT, Patent Nr. EP0929864B1*, March 2000.

[eK05a] eva Kühn. Coordination system. *Patent Nr. US6848109 - U.S.A*, January 2005.

[eK05b] Gerson Joskowicz eva Kühn, Johannes Riemer. XVSM (eXtensible Virtual Shared Memory) Architecture and Application. Technical report, TU-Vienna, E185/1, SBC-Group, 2005.

[FAH99] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1st edition, 1999.

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

[GdMFnG+99] José Ramón González de Mendívil, Federico Fariña, José Ramó Garitagoitia, Carlos F. Alastruey, and J. M. Bernabeu-Auban. A distributed deadlock resolution algorithm for the and model. *IEEE Trans. Parallel Distrib. Syst.*, 10(5):433–447, May 1999.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

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

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, December 1987.

[Gra78] J. Gray. Notes on data base operating systems. In R. Bayer, R. Graham, and G. Seegmüller, editors, *Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer Berlin / Heidelberg, 1978. 10.1007/3-540-08755-9_9.

[HDDG04] M.H. Haji, P.M. Dew, K. Djemame, and I. Gourlay. A snap-based community resource broker using a three-phase commit protocol. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 56–, 2004.

[ISO92] ISO. *ISO/IEC 9075:1992, Database Language SQL*. International Organization for Standardization, 1992.

[KN98] Eva Kühn and Georg Nozicka. Post-client/server coordination tools. In Wolfram Conen and Gustaf Neumann, editors, *Coordination Technology for Collaborative Applications*, volume 1364 of *Lecture Notes in Computer Science*, pages 231–253. Springer Berlin Heidelberg, 1998.

[Lam81] Butler Lampson. Chapter 11. atomic transactions. In D. Davies, E. Holler, E. Jensen, S. Kimbleton, B. Lampson, G. LeLann, K. Thurber, and R. Watson, editors, *Distributed Systems — Architecture and Implementation*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer Berlin / Heidelberg, 1981. 10.1007/3-540-10571-9_11.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[Lam02] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Lam06] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[LL93] Butler Lampson and David Lomet. A new presumed commit optimization for two phase commit. In *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, pages 630–630. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1993.

[Moi11] Izabela Moise. Efficient agreement protocols in asynchronous distributed systems. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 0:2022–2025, 2011.

[MOZ05] Dahlia Malkhi, Florin Oprea, and Lidong Zhou. Omega meets paxos: Leader election and stability without eventual timely links. In Pierre Fraigniaud, editor, *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 199–213. Springer Berlin Heidelberg, 2005.

[SBCM93] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimizations and tradeoffs in the commercial environment. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, pages 520–529, 1993.

[SS83] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed systems. *IEEE Transactions on Software Engineering*, pages 219–228, 1983.

[vHvHLP07] Frank van Harmelen, Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of Knowledge Representation*. Elsevier Science, San Diego, USA, 2007.

[WeKT00] Heidemarie Wernhart, eva Kühn, and Georg Trausmuth. The replicator coordination design pattern. *Future Generation Computer Systems*, 16(6):693 – 703, 2000.

[Zar12] Jan Zarnikov. Energy-efficient persistence for extensible virtual shared memory on the android operating system. Master's thesis, Institut für Computer Languages, Vienna University of Technology, 2012.

Web references

[1] IBM Corporation. How consistency is maintained. http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/index.jsp?topic=%2Fcom.ibm.mq.doc%2Fzc11310_.htm, October 2012. Accessed: 2013-01-15.

[2] Apache Software Foundation. JavaSpaces Service Specification. http://river.apache.org/doc/specs/html/js-spec.html. Accessed: 2013-04-10.

[3] The Apache Software Foundation. Apache River. http://river.apache.org/. Accessed: 2013-01-15.

[4] The Apache Software Foundation. Jini Transaction Specification. http://river.apache.org/doc/specs/html/txn-spec.html. Accessed: 2013-01-15.

[5] The Open Group. Distributed Transaction Processing: The XA Specification. http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf. Accessed: 2013-04-22.

[6] Microsoft. Usability in Software Design. http://msdn.microsoft.com/en-us/library/ms997577.aspx, October 2000. Accessed: 2013-04-13.

[7] OASIS. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, April 2007. Accessed: 2013-01-31.

[8] Oracle. Distributed Transaction Processing: XA. http://docs.oracle.com/cd/E11882_01/timesten.112/e21637/xa_dtp.htm. Accessed: 2013-01-15.

[9] Oracle. Java Transaction API (JTA). http://www.oracle.com/technetwork/java/javaee/jta/index.html. Accessed: 2013-04-22.

[10] GigaSpaces Technologies. Inside GigaSpaces XAP - Technical Overview and Value Proposition. http://www.gigaspaces.com/system/files/private/resource/InsideXAP-August4_2011.pdf+InsideXAP-August4_2011, August 2011. Accessed: 2013-01-11.