FAKULTÄT FÜR !NFORMATIK

Design and Implementation of MozartSpaces, the Java Reference Implementation of XVSM

Custom Coordinators, Transactions and XML protocol

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering / Internet Computing

eingereicht von

Christian Schreiber
Matrikelnummer 0325661

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuerin: A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn
Mitwirkung: Univ.-Ass. Dipl.-Ing. Richard Mordinyi

Wien, 08.09.2008 (Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Abstract

Designing and implementing distributed applications is a complex process. A lot of aspects, like scalability, availability, security, synchronisation and heterogeneous environments have to be considered. This thesis describes MozartSpaces, the Java implementation of XVSM (eXtensible Virtual Shared Memory). XVSM is an extensible, distributed, space based computing middle-ware which addresses a lot of the issues very well. It supports developers in creating distributed applications by offering a natural, p2p based abstraction over the underlying physical network. Instead of exchanging messages like in conventional systems, communication on top of XVSM is realised by shared data structures. This thesis deals especially with the realisation of the transaction management, the extensibility through custom coordination models and the platform independent XML protocol.

Kurzfassung

Planen und implementieren von verteilten Systemen ist ein komplexer Prozess. Sehr viele Aspekte wie Skalierbarkeit, Verfügbarkeit, Sicherheit, Synchronisation und heterogene Umgebungen müssen berücksicht werden. Diese Diplomarbeit beschreibt MozartSpaces, die Java Implementierung von XVSM (eXtensible Virtual Shared Memory). XVSM ist eine erweiterbare, verteilte, space based computing Middleware, welche viele der erwähnten Probleme sehr gut löst. Sie unterstützt Entwickler beim Erzeugen von verteilten Anwendungen durch eine natürliche, p2p basierte Abstraktion des zugrundeliegenden physischen Netzwerkes. Im Gegensatz zur herkömmlichen Nachrichten basierten Kommunikation erfolgt die XVSM basierte Kommunikation zwischen verteilten Anwendungen über gemeinsam benutzte Datenstrukturen. Diese Arbeit geht besonders auf die Realisierung des Transaktionmanagements, die Erweiterbarkeit durch benutzerdefinierte Koordinationsmodelle und das plattformunabhänge XML Protokoll ein.

Contents

1 Introduction 1

1.1 Extensible Virtual Shared Memory 2

1.1.1 Container and Entries 4

1.1.2 XVSM API 5

1.1.3 Aspects 7

2 Structure of the XVSM Kernel 9

2.1 The XVSM Run-time Model 9

2.2 Implementation of the run-time in MozartSpaces 11

2.2.1 Thread Pools 11

2.2.2 Internal Processing of Operations 12

2.2.3 Container Architecture 17

2.2.4 Container Management 23

2.2.5 Execution of Aspects 25

2.2.6 Transaction Management 25

2.2.7 Timeout Management 25

2.2.8 Remote Communication 27

3 Dynamic Kernel Configuration 30

3.1 Reconfigure during Run-time 30

3.2 Parameters 31

4 Exceptions 33

4.1 Exception Handling 33

4.2 Exception Description 34

4.3 ICapi Exceptions 35

5 Transactions 37

5.1 Transaction Management 37

5.2 Sub-transactions 38

5.3 Locking 39

5.3.1 Container Locking 40

5.3.2 Entry Locking 42

5.4 Transactional Container Operations 48

5.5 Transaction Life-cycle 48

5.5.1 Create Transaction 48

5.5.2 Commit / Rollback 49

6 Custom Coordinators **52**
6.1 API Description . . . 52
6.2 Predefined Coordinators . . . 56
6.2.1 Random . . . 56
6.2.2 FIFO . . . 57
6.2.3 LIFO . . . 57
6.2.4 Linda . . . 57
6.2.5 Vector . . . 58
6.2.6 Key . . . 59

7 XML Protocol **61**
7.1 Supported Data Types . . . 61
7.2 XML Serialisation of Properties . . . 61
7.3 XML Serialisation of Selectors . . . 62
7.4 XML Serialisation of Entries . . . 63
7.5 XML Serialisation of Coordinators . . . 64
7.6 Serialisation of Operations in XML . . . 65

8 Related Work **71**
8.1 JavaSpaces . . . 71
8.2 TSpaces . . . 72
8.3 Corso . . . 72
8.4 ActiveSpace . . . 72
8.5 XMLSpaces . . . 73
8.6 LIME . . . 73
8.7 LighTS . . . 73
8.8 Mars . . . 74
8.9 TuCSoN . . . 74

9 Implementing a Semantic Space **75**

10 Future Work **78**

11 Conclusion **79**

References **80**

A Interface Descriptions **I**

List of Figures

1 XVSM Core architecture. 2

2 XVSM P2P architecture. 3

3 Structure of an XVSM container. 8

4 The XVSM Runtime Architecture. 9

5 Architecture of an Container in MozartSpaces. 17

6 Sequence diagram illustrating the successful execution of an operation within a container. 24

7 Class diagram of the classes involved in the remote communication. 28

8 Inheritance hierarchy of the MozartSpaces exceptions. 33

9 Sequence diagram of a successful commit operation. 50

10 Inheritance hierarchy of the coordinator interfaces 52

11 Internal data structure of the key coordinator. 59

12 Container structure of the semantic space 76

List of Tables

1 Description of the fields of AspectTask 13
2 Description of the fields of ContainerTask 14
3 Description of the fields of OperationTask 15
4 Description of the fields of TransactionTask 16
5 Reasons for blocking operations and for waking them up 19
6 Possible exception of the ICapi interface 36
7 Actions to be taken on commit or rollback 47
8 Mapping TripCom components to XVSM 75
9 ITransactionLayer interface III
10 ContainerEngine interface IV
11 ContainerManager interface VI
12 ConfigurationManager interface VI

List of Listings

1 Interface of Task.java 12
2 IContainer interface 17
3 ITransactionLayer interface 20
4 initTransaction method 20
5 IContainerEngine interface 21
6 read operation pseudo code 22
7 shift operation pseudo code 22
8 Interface of the Container Manager 24
9 method which is used to calculate the remaining timeout 26
10 ConfigurationManager interface 30
11 Interface of the TransactionManager 38
12 Implementation of acquireLock method 40
13 Implementation of commitLocks method 41
14 Implementation of rollbackLocks method 42
15 Code for creating a new transaction identifier 49
16 XML representation of properties 62
17 XML representation of selectors 62
18 XML representation of entries 63
19 XML representation of entries 64
20 XML representation of a create container operation 65
21 XML representation of a destroy container operation 66
22 XML representation of a write operation 66
23 XML representation of a read operation 66
24 XML representation of a destroy operation 67
25 XML representation of a shift operation 67
26 XML representation of a create transaction operation 68
27 XML representation of a commit operation 68
28 XML representation of a rollback operation 68
29 XML representation of a clear space operation 68
30 XML representation of a shutdown operation 69
31 XML representation of an add aspect operation 69
32 XML representation of a remove aspect operation 70

List of Abbreviations

ACID .... Atomicity, Consistency, Isolation, Durability
capi .... Core API
CREF .... Container Reference
DAO .... Data Access Object
DHT .... Distributed Hash Map
FIFO .... first in first out
IPoint .... Interception Point
JMS .... Java Message Service
LIFO .... last in first out
P2P .... peer to peer
RDF .... Resource Description Framework
RMI .... Remote method invocation
SBC .... Space based computing
SEDA .... Staged Event Driven Architecture
SQL .... Structured Query Language
TCP .... Transport Control Protocol
TCP .... Transport control protocol
UML .... Unified Modeling Language
URL .... Uniform Resource Locator
XML .... Extensible markup language
XTM .... Extensible tuple model
XVSM .... Extensible Virtual Shared Memory

1 Introduction

The design and creation of distributed application is a complex and error-prone process. A lot of aspects have to be considered: are devices/applications available when they are needed, synchronisation of concurrent applications, network failures, heterogeneous environments, special behaviour of different operating systems, security and so on. A lot of principles and middle-wares [45, 44, 23] have been developed to support developers and system designers. Systems like message passing and remote procedure call middle-wares abstract from the complexity of the underlying network and transport protocol but the communication partners have to know each other. If one component of a distributed system wants to distribute information to all other components it has to send this information to each one separately (even if they are not interested in this information at this moment, because it is not possible for the application to retrieve the information again later). If an application is not available it will miss this information. In addition, this leads to a big amount of data which is sent through the network. In order to deal with components which are not always available, message queues and publish/subscribe systems [46] have been developed which store a message until the component fetches the information. Unless publish/subscribe systems are used which allow multiple receivers for one message, it is necessary that the producer knows who is interested in certain information. If a distributed system has a lot of participants or the number is varying very often, the deployment of such an application can get very complicated.

Many of the above described problems are very well addressed by space base computing (SBC) middle-wares. They provide a shared data space in which application can store and from which they can retrieve information [31]. Applications using a space based computing middle-ware for communication do not communicate directly with each other. Instead, they use data objects (these are often called tuples) for communication. It is not necessary for application to be on-line all the time since information is stored within the space. Producers do not have to deal with the list of receivers. They only have to write the information into the space and the receivers pro-actively fetch the information when they need it.

This thesis describes the implementation of a Java based distributed space called MozartSpaces. The main focus of this thesis is on transaction mechanisms, the extensibility of the coordination structures, and the XML based protocol which is used for communication.

This thesis is structured as follows: in Section 1.1 the features of the implemented space are described, Section 2 describe the overall structure of the implementation, Section 3 deals with the configuration options of the space, Section 4 describes the exception handling, Section 5 describes the implementation of the transaction mechanism, Sec-

tion 6 describes how to implement custom coordination strategies, Section 7 describes the communication protocol and in Section 9 an application is described using some of the features of the implemented space.

At the time of writing, the implementation of MozartSpaces consisted of 111 Java classes with about 19700 lines of code. The test environment (mainly unit tests) contains about 11000 lines of code in 71 classes.

1.1 Extensible Virtual Shared Memory

XVSM stands for extensible virtual shared memory [34, 33, 32]. It is a space based computing middleware which defines a Linda [28, 29] tuple space based extension. XVSM realises the concept of an distributed *object space* which can be accessed concurrently via a network.

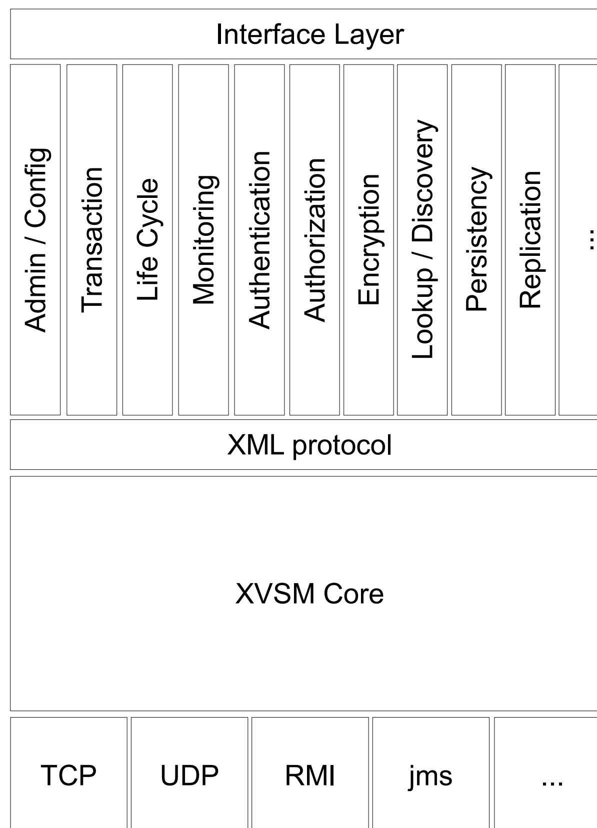

Figure 1: XVSM Core architecture.

The architecture of an XVSM kernel is illustrated in Figure 1. The main part of the kernel is the “XVSM Core” which implements the business logic and therefore realises the main functionality of the space. It can be accessed using a standardised XML protocol (see Section 7). This protocol can be transported using several transport technologies (see Section 2.2.8) and standard messaging protocols. These could be a simple TCP

based communication, remote method invocation (RMI), Java Message Service (JMS) or much more complex communication mechanisms and standards. These protocols can be customised by a user and it is possible to support multiple transport protocol simultaneously. On top of the XML protocol so called functional profiles can be plugged to the core. The aim is to provide the possibility for extensions like persistency, distributed transactions, replication, security, and so on. The architecture comprises language bindings on top of the XML protocol which can be customised and optimised for specific application requirements.

The XVSM Core implements pessimistic transactions (see Section 5) with ACID properties. A fine-grained locking mechanism is used which allows concurrent transactions. Based upon this XVSM Core feature support, additional transaction-function profiles for e.g. two phase commit or distributed transactions can be implemented.

XVSM is designed to run in a P2P manner where each client has an XVSM instance within its process or in a server/client structure where clients only send and receive XML messages from/to a server instance.

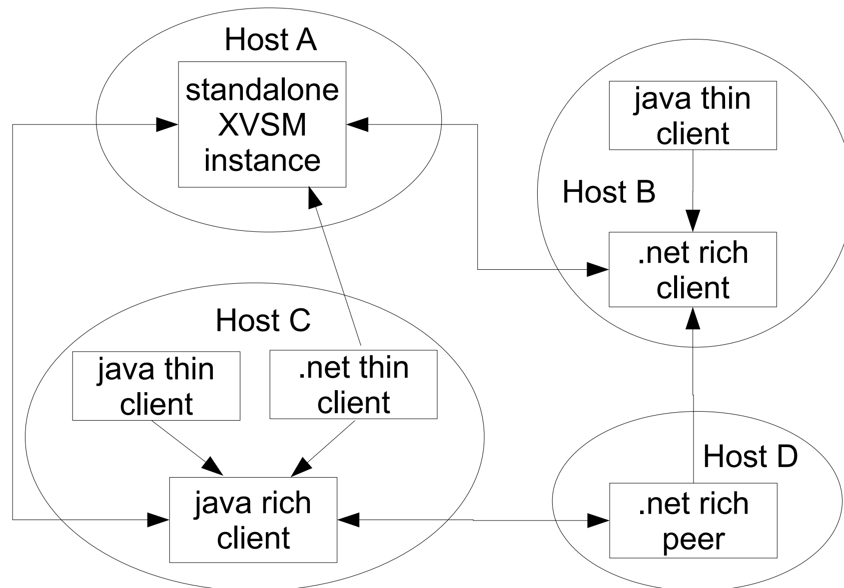

Figure 2: XVSM P2P architecture.

As depicted in Figure 2 XVSM can be either used as standalone instance which can be accessed by other instances or by thin clients[1] or be used in an embedded mode running within the same process as the client (i.e., rich peer). Rich peers can as well be accessed from other peers or processes on the same peer. Currently a Java [5] (called MozartSpaces) [10] and a .net (called XcoSpaces) [15] implementation of the XVSM run-time exist. In

[1]Thin clients are capable of sending and receiving XVSM XML messages (see Section 7) but they do not have an embedded space instance running. Clients with an embedded space instance are called rich peers in XVSM terminology.

order to be independent of the platform and the used programming language an XML (see Section 7) is used for communication between instances.

1.1.1 Container and Entries

The basic concept of XVSM is a container. This is a programmable object in the space which can store and administrate so called data entries. A container can be addressed by its container reference URL ("CREF") so that a container becomes accessible like any other resource in the Internet. In addition to the CREF a container can have an optional name to support look-up mechanisms. Since a container stores entries like in the Linda model it can be considered as a (named) sub-space (but a container can not have sub-containers). Another important characteristics of a container is that it can be bounded or unbounded: Bounded means that within a container there is a limited number of places for entries. Entries have no identity and can either be structured Linda tuples (cf. database records), any basic data type (i.e., Character, Integer, Boolean, Long, Short, Float, Double), XML data, RDF data, or any other complex data structure (see Section 6).

Entries containing a basic data type are called *atomic* since they do not have an inner structure. A tuple can contain these atomic entries and again tuples. An entry can be tagged with meta information, called *selector* information, before it is stored within a container. This meta information can be a priority, the position within the container, an unique key, a label or any other user defined value. It is used by the XVSM Core to store and to manage the inserted entries. Furthermore, selectors can be used by a user to query entries.

A container only stores the data, the coordination and administration of the entries is done by so called *coordinators*. The purpose of a coordinator is to realise different coordination models, for example an order of entries, a prioritization between entries, or to access entries based on certain meta information (i.e., key). These coordinators can be extended and/or created by users to fit the needs of the application (see Section 6). Per default XVSM supports the following coordination models:

FIFO First On First Out, the realisation of a queue.

LIFO Last in First Out, the realisation of a stack.

Random this coordinator has no order, i.e., entries are returned randomly.

Key entries can be accessed using an unique key which has to be provided during the read/write operation.

Vector entries are stored in a list and can be access via their positions.

Linda realises the template matching (i.e., query by example) introduced by the Linda programming model [28, 29] .

Each coordinator has a corresponding selector (representing required meta information) which is used to store and query entries into/from the container. Coordinators interpret their corresponding selectors and store references to the entries in a way which is optimised for the purpose of the coordinator. For example, to store an entry in the key coordinator the key has to be passed to the coordinator when the entry is written to the container. This is realised using a key selector which contains this key. The base coordination type of a container is *random*. This means that every XVSM container can be queried using random selectors independently of the other supported coordinators. A random selector can always be used to retrieve entries within a container.

In XVSM it is distinguished between `implicit` and `explicit` coordinators. A coordinator is implicit if it has a view over all entries, this means, it knows about all entries in the container. These coordinators are called implicit because, if an entry is written to the container, the entry does not have to provide a selector in order to be stored within the coordinator (but it could). FIFO, LIFO and RANDOM are examples for implicit coordinators. Explicit coordinators are those which require additional meta information (in form of a selector) for storing the entries. This meta information has to be provided by a user who writes the entry, otherwise the entry will not be stored within the coordinator. Key and Vector are examples for explicit coordinators because they require the key which shall be used respectively the position at which the entry shall be added.

Additionally, every XVSM Container has a meta container storing information about the container. The meta container can be addressed by appending "/meta" to the container reference. Currently, the maximum size of the container, the current size of the container, the list of coordination types supported by the container and the container name (if available) can be read from the meta container.

1.1.2 XVSM API

The XVSM API (respectively the XML protocol) provides methods to *create* and *destroy* Containers. The create method takes the container name, the maximum container size and a list of coordinators which shall be supported by the container as optional arguments. The core returns the container reference which can be used to address the new container. The destroy container method requires the reference of the container which shall be destroyed. When a container is destroyed all entries within this container are destroyed as well.

According to the Linda model, the basic operations on a container are *write* (insertion of entries), *read* (read of entries) and *take* (consuming read of entries). XVSM supports

bulk operations, which allow the writing and reading of multiple entries within one operation. A bulk write operation can be used to write multiple entries at once into one container and a bulk read/take to get multiple entries from the container (the amount can be specified). Read and take offer blocking behaviour if no suitable entries are in the container. A timeout can be given which specifies the minimum time in milliseconds the method shall try to get the entries [41]. If the timeout is 0 the operation is tried exactly once and will not block if it can not be fulfilled. An infinite timeout is supported as well. Since a container can be bounded the write operation can block if the container has no free space for the entry or an entry with the same meta information (position within a vector coordinator or key in a key coordinator) already exists (this is determined by the used coordinators (see Section 6). The write method blocks until there is enough free space to write the entries or the timeout expires. In order to provide a non-blocking write operation the *shift* operation has been added to the XVSM API. This operation removes as many entries as necessary to have enough free space before writing the new entries. In addition a *destroy* method exists which behaves exactly as take but does not return the removed entry.

An XVSM read, take and destroy operation can have more than one selector. In this case, the selectors are executed in the same sequence as they where provided by the user. The result of the preceding selection is the input for the following one. Since the number of entries which shall be removed/returned is a parameter of the selector, the preceding selector can be used to decrease the number of input entries for the following selector. For example if the entry which has been written fifth of a container which uses a FIFO and a LIFO coordination shall be selected, a user could first select five entries using a FIFO selector and afterwards one entry using a LIFO selector.

Additionally, methods exist to *create*, *commit* and *rollback* transactions and to *add* and *remove* aspects. The create Transaction operation accepts a timeout which defines the life span of the transaction. When this timeout expires before the transaction has been committed or rolled back the transaction is rolled back by the core (and therefore it is invalid afterwards). The answer to a create transaction operation is a transaction reference which has to be passed to all operations which shall be executed under this transaction.

Every XVSM operation, except those for transaction and aspect management, accept a transaction reference as optional parameter. If no transaction is provided the operation is executed using an implicit transaction, i.e., a transaction is automatically created for the execution of this operation.

In addition to the parameters of the XVSM operations a so called *aspect context* can be passed with every operation. This can be used by a client to communicate with the XVSM aspects which are explained in Section 1.1.3. The aspect context is a key value

pair which has a string as key and an entry (or tuple) as value.

A request to an XVSM core is not answered by the core with an answer message directly. Instead, every request has to define an answer container in which the answer shall be written. The core writes the answer in form of an entry into this container and the requesting peer can retrieve the answer from this container. This container does not have to be a real XVSM container. In the case of a thin client this answer container can be seen as a request identifier which can be used to pass the answer to the correct request. This feature enables a user to decide where the answer to requests shall be sent. For example, if an operations takes a long time to execute, a user can go offline and fetch the answer later from the answer container if the answer container is located on an XVSM instance which is always available.

In order to simplify the testing process of an XVSM instance or an application using XVSM, methods for remotely shutting down an instance and to completely clear the content of the space are provided.

1.1.3 Aspects

In XVSM every API operation can be intercepted and a user can inject code which shall be performed before (PRE) or/and after (POST) the intercepted operation is executed. This injected code is called *aspect* in XVSM. The idea of aspects is inherited from the aspect oriented programming paradigm. Aspects can be added to containers (called local aspects) and to the whole space (called global aspects). A local aspect can intercept all operations executed on the container and a global aspect can intercept the container operation on all containers plus the API operations which are defined on the space (i.e., create/destroy Container and Transaction related operations). As described in [41], aspects are used to realise security (authorisation and authentication) and the implementation of a highly customisable notification mechanism. It is possible that multiple aspects are added to the same container. In this case, the aspects are executed sequentially in the same order in which they where added.

An XVSM aspect can manipulate the execution of the operation which triggered it. This is realised by return values which are provided by the aspect after the execution. The XVSM core reads the results and manipulates the execution of the operation accordingly. The following return values are supported:

OK Everything is fine, the execution of the operation proceeds normally.

SKIP The operation is not performed on the container or the spaces, the POST aspects are executed immediately (following PRE aspects are not executed). This return value is only supported for PRE aspects.

NOTOK The execution of the operation is stopped and the transaction which was used for the transaction is rolled back. It can be used by a security aspects to deny a operation if the user has not adequate access rights on the container or the space.

RESCHEDULE The execution of the operation is stopped and it will be rescheduled for a later execution. This can be used to delay the execution of an operation until an external event occurs.

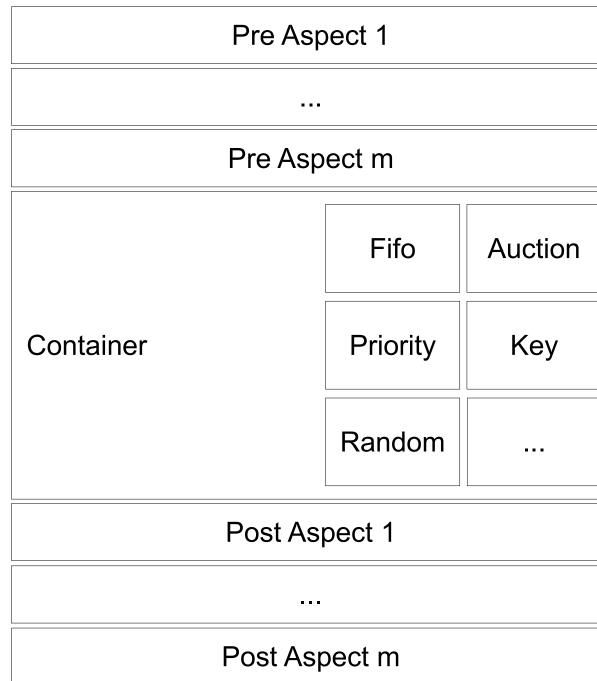

Figure 3: Structure of an XVSM container.

Figure 3 depicts all components of an XVSM container. The central part of a container is the implementation of the container's business logic which handles the storage of the entries and coordinates the coordinators. An operation has to pass the PRE-aspects before it is executed on the container. If all PRE-aspects return OK, the container implementations interprets the selectors of the operations and executes the operation. Afterwards, all POST-aspects are executed. Depending on the the result of the POST-aspects the operations result is returned to the requesting peer which executed it or the operation is rolled back.

2 Structure of the XVSM Kernel

In this section a description of the XVSM run-time architecture is given. First, in section 2.1, the architecture model on which the Java implementation, MozartSpaces, is based is described. In section 2.2 the actual implementation of MozartSpaces is explained and how the architecture model has been implemented. Since XVSM implementation in .net [43] and in Java exist the idea of the model is to have a common abstract architecture description of both implementations.

2.1 The XVSM Run-time Model

The XVSM run time architecture combines mechanisms derived from SEDA [49],[50] and XVSM itself. In SEDA, the execution of an operation is realised by putting the input information into an event queue. A bounded amount of threads continuously checks if a new event is available. One of the threads takes the new event from the event queue and executes it. The internal execution of operations is handled within the XVSM kernel in the same way besides that XVSM containers are used as event queues. Currently, these containers manage the events (respectively the entries) in a FIFO manner. In later versions a priority based coordination could be supported which would enable the prioritisation of API operations within the kernel.

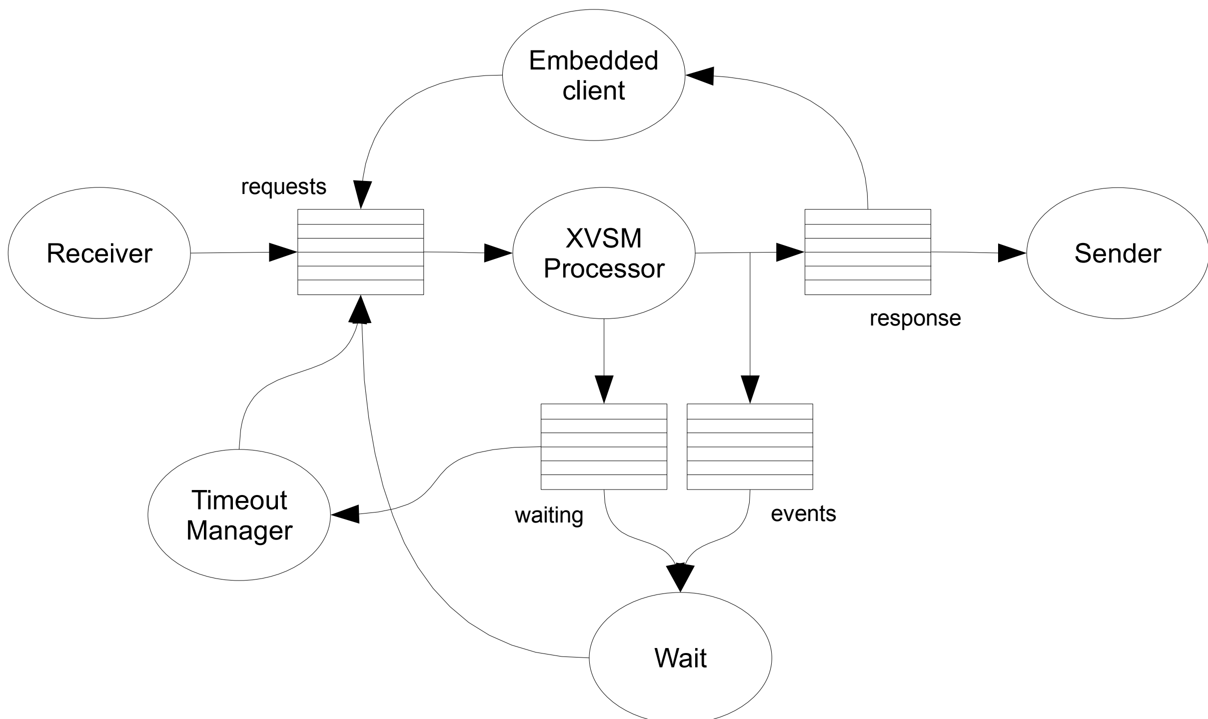

Figure 4: The XVSM Runtime Architecture.

Figure 4 depicts the internal processing of API operations. The ellipses represent

thread pools which are used to execute the operations and the rectangles represent the XVSM containers. The arrows indicate the message flow between the components. A thread pool combined with the containers from which it reads the input can be considered as a SEDA stage. In this design it is also necessary that one thread pool has multiple containers as input queue. The communication between the stages takes place by means of exchanging tuples. Every thread within the thread pool executes a blocking take (with infinite timeout) on the container containing the input. When a new request is written to the container, one of the threads gets the request and executes it. Afterwards, the thread writes the request (updated with the result of the computation) into the input container of the next stage. Using this run-time architecture XVSM itself is implemented based upon ideas and mechanisms of Space Based Computing middle-wares. In the following all parts of the model are explained in more detail.

Receiver. This component handles the receiving of operations. It has to coordinate the supported transport protocols and takes care of marshaling incoming request into an internal representation. When a new operation arrives, the receiver component writes the operation into the "requests" container which is the input container of the XVSM Processor.

XVSM Processor. This component does the execution of operations within the core. It has to administrate the containers and transactions and has to make sure that API operations are executed correctly. Therefore, this component is the central part of an XVSM core instance. When the execution of an operation has to be blocked this component writes it into the "waiting" container otherwise it is written into the "response" container. Every operation which is written to the latter container is also written to the "events" container. The waiting and the events container contain the input for the "wait" component and the response container stores the input for the "sender" component.

Sender. This component is responsible to send the result of the operation to the correct receiver. Since the result of an operation is always written to an container the sender component first determines to which XVSM instance the result has to be sent. Afterwards, it creates an XVSM write operation request containing the answer and sends the request.

Wait. This component checks if it is necessary to wake up blocking operations. This is realised by interpreting the operations written to the events container and comparing them to the blocking operations (stored in the waiting container). If an issued operation could cause a blocking operation to be fulfilled the "wait" component

removes this blocking operation from the wait container and writes it into the request container. For example, if a blocking read is in the wait container and a client wrote an entry into the container on which the blocking read was initially issued, this component removes the blocking read and writes it into the request container because it is possible that the read operation now can be fulfilled.

Timeout Manager. This component deals with the timeout management. It has to make sure that the timeout of operations and transactions are checked periodically and that they are removed from the core if their timeout expired.

Embedded client. This is actually not a component of the XVSM core. An embedded client is an XVSM client which runs in the same process as the XVSM kernel runtime. This sort of client does not have to use the receiver and the sender component. Instead, it can write its requests directly to the requests container and is able to wait for the corresponding response in the response container. Using this mechanism it is possible for a client to read from its answer container (if it is in the embedded core) without the need of offering another answer container which would cause an infinite loop.

2.2 Implementation of the run-time in MozartSpaces

In section 2.1 the XVSM run-time model has been described. In this section the actual implementation which is based on this model is explained. In order to better fit the Java programming language and to optimise some of the operations the MozartSpaces architecture is not completely equal to the XVSM run-time model. Differences are explained in the following sections.

2.2.1 Thread Pools

MozartSpaces is implemented using bounded thread pools to decrease the memory and dispatching overhead of threads. As described above, the implementation is based on SEDA principles which enables a constant throughput even when the rate of concurrent operations increases. In MozartSpaces two different types of thread pool implementations from the Java 5 default API are used:

- `ExecutorService`, this interface has been introduced to Java in version 1.5. It offers methods accepting an instance of a `Runnable` or `Callable` which will be executed by one thread of the thread pool. To instantiate an object implementing this interface, the factory method `Executors.newCachedThreadPool()`[2] has been

[2] `http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Executors.html#newCachedThreadPool()`

used. This factory method returns a thread pool implementation which creates new threads as needed, i.e., if there are task which have to be executed and no idle threads are available. If a thread is idle for more than 60 seconds it is destroyed. This thread pool is used to execute the operations, see Section 2.2.2 for a detailed description.

- `ScheduledThreadPoolExecutor`[3], this thread pool implementation accepts also `Runnable` or `Callable` implementations, but instead of executing an task once it is executed periodically after a given delay. In the default configuration of the delay is set to 500 milliseconds, but this can be configured using the configuration file (see Section 3). In MozartSpaces this thread pool is used for timeout handling (see Section 2.2.7).

MozartSpaces currently does not use XVSM containers as queues for the thread pools as described in section 2.1. Instead, the default thread pool implementations of the Java API are used, but MozartSpaces uses wrapper classes (called `EventProcessingPool` and `TimeoutSchedulerPool`) to access the thread pools which makes it simple to replace or extend the thread pool implementations by manipulating the wrapper classes.

2.2.2 Internal Processing of Operations

In MozartSpaces all operations which are executed within the core are represented by an instance of the class `Task`. This class extends the Java interface `Runnable` and therefore can be used to be executed by the thread pools. The interface of the class is depicted in Listing 1.

```
1  final void setResult(Object o);
2  final Object readResult();
3  Object getResult();
4  final void waitUntilFinished();
5  boolean hasFinished();
6  URI getAnswerToContainer();
7  void setAnswerToContainer(URI answerToContainer);
8  Properties getAspectContext();
9  void setAspectContext(Properties aspectContext);
```

Listing 1: Interface of Task.java

Task.java provides methods to block a thread until the result of the task is available. This is internally realised with `wait`[4] and `notify`[5]. The methods `readResult` and `waitUntilFinished` block until the result is set with `setResult`. Additionally, the methods `getResult` and `hasFinished` are provided which return immediately and do

[3]http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html
[4]http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html#wait(long)
[5]http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Object.html#notify()

not block even if the result is not available. Since every XVSM operation has an answer container and an aspect context, operations for setting and getting them are provided. `setResult`, `readResult` and `waitUntilFinished` are marked as final because they are used for waking up client threads if they are running embedded, therefore, they should not be overwritten in subclasses.

There are seven classes extending the abstract class Task in MozartSpaces which are summarised in tables 1-4. The classes implement the business logic in the `run()` method which will be invoked by the thread pool. This business logic is explained in greater detail below. After the task has been executed successfully the task itself has to set the result of the execution using `setResult(Object o)` (if a method returns `void` the result has to be set to `VoidEntry` which is special entry indicating that the result of an operation is void).

AspectTask : implements the logic for adding and removing aspects.

Field	Description
AspectTaskType type	enumeration which indicates if an aspect shall be added or removed.
IAspect aspect	the aspect which shall be added. This field is `null` if an aspect is removed.
List<IPoint> points	list of interception points to which the aspect shall be added. This field is `null` if an aspect is removed.
ContainerRef cref	the container to which the aspect shall be added. This field is `null` if the aspect is an global aspect.

Table 1: Description of the fields of AspectTask

First, it is checked whether a PRE-aspect shall be added or removed. In both cases the global aspects which are registered for this operation are then executed. If all aspects are answered with OK, a reference to the internal container implementation (see Section 2.2.3) is obtained from the container manager (see Section 2.2.4). If the provided container reference is valid the aspect is added to respectively removed from the container. Finally, the POST-aspects are executed.

ClearTask : internal representation of the clear space API call. This task does not define additional fields.

The task starts with obtaining a reference to the TransactionManager (see Section 2.2.6) which is used to get a list of all currently active transactions and rolls them back. Afterwards, all containers are removed by using the ContainerManager (see Section

2.2.4). Finally, the TransactionManager, ContainerManager and the ConfigurationManager (see Section 3.1) singleton instances are removed.

ContainerTask : implements the logic for creating, deleting and getting a container.

Field	Description
ContainerTaskType type	enumeration indicating if a container shall be created, destroyed or a named container shall be looked up.
ContainerRef cref	the container reference of the container which shall be destroyed. This field is `null` if look-up or a create operation is executed.
boolean meta	indicates if the meta container shall be returned.
String name	this field is set to the name of the container if a named container is created or looked up, otherwise this field is `null`.
int size	the size of the container. If the new container shall be unbounded this field has to be set to the constant `IContainer.INFINITE_SIZE`.
Transaction tx	the transaction which shall be used to execute the operation. If an implicit transaction shall be used this field is set to `null`.
ICoordinator[] coordinators	the array contains the coordination types which shall be supported by the new container. If a look-up or a delete operation is executed this field is set to `null`.

Table 2: Description of the fields of ContainerTask

First a reference to the container manager is obtained and all global PRE-aspects which are registered for create-, delete- or get container are executed. If every aspect answers with OK the operation is executed using the ContainerManager by invoking the corresponding method. In case of create- and getContainer the container reference which is returned from the container manager is set as result. The result of delete container is void. Finally, the POST-aspects are executed.

OperationTask : this task represents an operation executed on a container.

Field	Description
OperationTaskType type	enumeration indicating the type of operation (i.e., read, write, take, shift, destroy).

long timeout	the timeout of this operation. If a switch operation is executed this field is ignored.
long lastTimeoutUpdate	this field is required for the internal timeout management. It contains the last time when the timeout scheduler checked the timeout of this task.
ContainerRef cref	the container on which the operation shall be executed.
Transaction tx	the transaction under which the operation shall be executed. If an implicit transaction shall be used this field is set to `null`.
List<Selector> selectors	the selectors used for the operation.
List<Entry> entries	the entries used in the operation. This field contains the entries which shall be written in case of shift or write operation. If a read or take operation is executed it contains the result of the operation.
List<Entry> deleted	entries which were deleted during the execution of the operation. This field is used for take, delete and shift operations.
int retrycount	this integer indicates the number or attempts which have been taken to execute the operation (i.e., when the operation blocked).

Table 3: Description of the fields of OperationTask

In this task the first thing which is done is to obtain a link to the transaction object using the transaction manager. This is necessary because only the transaction identifier is transported over the network but for the internal execution of the operation the corresponding transaction object is required. Afterwards, the container manager is used to get the implementation of the container which belongs to the container reference and all global PRE-aspects are executed using the GlobalAspectManager (see [41]). Now, the operation is executed on the container. Finally, all POST-aspects are executed and the result of the operation is set.

ReplyTask : this task handles the writing of the answer to the according answer container. It does not define additional fields.

This task first checks which kind of result should be sent. If it is a list, the result corresponds to the result of a read or take operation, these entries will be written into the answer container, therefore all selectors and locks are removed and all VoidEntries are removed from the list. If after the removing of the VoidEntries the list is empty, the result

of the read operation is an empty list which is indicated by a VoidEntry. If the result is an instance of the Java class Throwable, an exception occurred during the execution. Therefore an ExceptionEntry is created which contains the name of the Exception and the description of the exception. After the type of the result is determined, the task gets the answer container reference and gets a reference to the TransportSender (see Section 2.2.8). This TransportSender is then used for sending the answer to the correct core instance.

ShutdownTask : implements the logic for shutting down a MozartSpaces instance. This task does not define additional fields.

As for all operations which can be intercepted, the task first executes all PRE-aspects. If all aspects answer with OK the EventProcessingPool, the TimeOutSchedularPool, the ReplySenderPool (see Section 2.2.8) and the TransportHandler instances are shut down. Afterwards the result has to be sent manually to the answer container because all internal core mechanisms are shut down and therefore can not be used. Additionally, there is no POST-shutdown aspect because it is not possible to do anything after shutting down the core instance.

TransactionTask : implements the logic for creating, committing and rolling back of a transaction.

Field	Description
TransactionTaskType type	enumeration indicating the type of the operation (i.e., commit, rollback, create).
Transaction tx	the transaction which shall be rolled back or committed. In case of a create transaction operation this field is used to store the new transaction.
long timeout	the timeout of the new transaction. If the transaction shall have an infinite transaction this field has to be set to the constant `ICapi.INFINITE`.

Table 4: Description of the fields of TransactionTask

As in the OperationTask the first thing done in the TransactionTask is to obtain the transaction object which belongs to the transaction reference using the transaction manager (see Section 5.1). Afterwards, all PRE-aspects are executed. The actual commit, rollback or create of the transaction is done in the transaction manager. This is explained in Section 5 in greater detail. After the execution of the operation the POST-aspects are executed and the result is set. For commit and rollback VoidEntry is set as result and

for create the transaction reference is set as result.

2.2.3 Container Architecture

A container is composed of three layers having separated functionality. These layer are the *Blocking Layer*, *Transaction Layer* and the *Container Engine*. Figure 5 illustrates the architecture of a container in XVSM and the position of the layers. The Container Engine also has to manage the coordinators which are as well depicted in the figure (C1,. . . , C4). In the following every layer is described in greater detail.

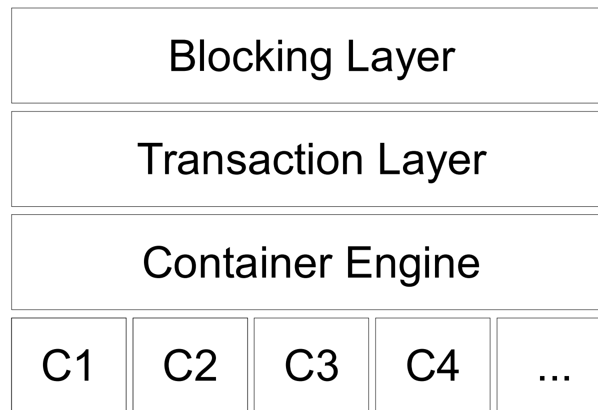

Figure 5: Architecture of an Container in MozartSpaces.

Blocking Layer This layer implements the `IContainer` interface. This interface is used by other components of the MozartSpaces core to communicate with the Container. The methods of the `IContainer` interface are shown in Listing 2. The most important method is `execute` which requires an `OperationTask` as parameter. It is used to execute API operations on the container. The methods `commit` and `rollback` are invoked when a transaction has to be committed respectively rolled back. Additionally, getter and setter methods for the container reference, the aspects, and the coordinators, and a method for getting the currently available number of entries are provided. In order to enable a container instance to clean up before it is destroyed the interface defines the method `destroy`. The exceptions which are declared to be thrown are explained in Section 4. If a user wants to create its own Container implementation only this interface has to be implemented. It is not necessary to provide the same container architecture as the one currently implemented.

```
1  Object execute(OperationTask task) throws ContainerFullExcpetion,
2      CountNotMetException, TransactionLockException,AspectRescheduleException,
           AspectNotOkException;
3  void commit(Transaction txn) throws TransactionLockException,
4      InvalidTransactionException, AspectRescheduleException,
5      AspectNotOkException;
```

```
6   void rollback(Transaction txn) throws InvalidTransactionException,
7       TransactionLockException, AspectRescheduleException,
8       AspectNotOkException;
9   ContainerRef getCref();
10  void setCref(ContainerRef cref);
11  void updateTimeouts();
12  void addAspects(List<IPoint> p, IAspect aspect, Properties aspectProperties);
13  void removeAspect(IPoint p, IAspect aspect, Properties aspectContext);
14  void addCoordinator(Class<? extends Selector> s, ICoordinator c);
15  List<ICoordinator> getCoordinators();
16  void destroy();
17  int currentSize();
```

Listing 2: IContainer interface

The blocking layer handles the blocking of operations if this is necessary. This is a small divergence to the run-time architecture described in Section 2.1. Instead of having a central component which has to manage blocking operations, each container manages its blocking operations on its own. When the `execute` method is invoked, the blocking layer analyses the task and executes the operation on the container. If the operation was not successful and has to be blocked, the blocking layer stores the task in an internal data structure. The storage mechanism for the blocking operation is accessed using an DAO (Data Access Object) [17] interface. The current implementation uses a Java HashMap to store the task. Other implementations, for example using an XVSM container or a database, can be added by implementing the DAO interface. If later on an operation is issued which could cause this operation to be fulfilled the blocking layer removes the operation from its internal data structure and passes it to the thread pool which executes the tasks.

Table 5 summarises the reasons why an operation can be sent to a blocking state. Basically, this can have two reasons: 1) entries or 2) another active transaction. Since MozartSpaces supports plugable coordination types, the distinction between the two reasons can not be done because the coordinators determine if an operation can be fulfilled or not.

For instance, it is possible that an operation is sent to the blocking state because of another transaction but it is not necessary to commit the transaction in order to successfully execute this operation. Another operation could be sufficient: A read operation using a random selector is blocked because all available entries are used by another transaction. If one entry is written to the container the read operation can be completed even when the transaction using the other entries has not been ended.

Besides blocking of operations the thread synchronisation is done in the blocking layer. This is realised by using the Java `ReadWriteLock`[6] interface. The interface defines two

[6] `http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/locks/ReadWriteLock.html`

operation	reason for blocking	reason for waking up
read take delete	there are not enough entries in the space to fulfil the selector	new entries are written to the container
	the available entries are used by another transaction	the transaction which used the entries is committed or rolled back
write	there is not enough free space available in the container to write the entries or an entry with the same meta information exists already	entries are deleted (taken, destroyed) from the container
	entries are marked as deleted but the transaction under which they where deleted has not been committed (see Section 5 for a description of the transaction mechanism of MozartSpaces)	the transaction is committed
	entries have been written but the transaction under which they where written has not been committed	the transaction is rolled back
shift	entries which have to be shifted from the container are locked by another transaction	the transaction is committed or rolled back

Table 5: Reasons for blocking operations and for waking them up

methods, `readLock()` and `writeLock()` which can be used to acquire a read respectively a write lock. Multiple read locks are emitted but only one thread can get a write lock. Every operation acquires the required lock before it is executed. Only if the lock can be retrieved the operation is executed. The `write`, `take`, `delete` and `shift` operations try to get a write lock and the `read` operation tries to acquire a read lock. When the operation finishes (either normally or abnormally (i.e. an exception is thrown)) the lock is released. Therefore, only read operations are allowed to be executed simultaneously, write operations are executed serialised. This simplifies the development of the other parts of the container, especially the coordinators, since there is always only one write operation active within a container.

Transaction Layer This is the second layer and has to handle the transaction management within a container. The interface of this layer is shown in Listing 3. Just like the `IContainer` interface this interface offers methods for the commit and rollback of transactions, adding and removing of aspects, for adding and getting coordinators and the container reference and for getting the maximum and the current size of the container. Instead of having one method for executing API operations this interface provides a method for each API operation. A detailed description of all parameters can be is given in table 9 in appendix A and an explanation of the exceptions is given in Section 4.

```
void commit(Transaction tx) throws TransactionLockException, InvalidTransactionException
    , AspectRescheduleException, AspectNotOkException;
int getSize();
int currentSize();
List<Entry> read(Transaction tx, List<Selector> selectors, int retrycount,
Properties aspectContext) throws NoSuchCoordinationTypeException, CountNotMetException,
    InvalidTransactionException, TransactionLockException, AspectRescheduleException,
    AspectNotOkException;
void rollback(Transaction tx) throws InvalidTransactionException,
    TransactionLockException, AspectRescheduleException, AspectNotOkException;
List<Entry> shift(List<Entry> entries, Transaction tx, Properties aspectContext) throws
    NoSuchCoordinationTypeException, InvalidTransactionException,
    TransactionLockException, AspectRescheduleException, AspectNotOkException;
List<Entry> take(boolean isDelete, Transaction tx, List<Selector> selectors, int
    retrycount, Properties aspectContext) throws NoSuchCoordinationTypeException,
    CountNotMetException, InvalidTransactionException, TransactionLockException,
    AspectRescheduleException, AspectNotOkException;
void write(List<Entry> entries, Transaction tx, int retrycount, Properties aspectContext
    ) throws ContainerFullExcpetion, NoSuchCoordinationTypeException,
    InvalidTransactionException, TransactionLockException, AspectRescheduleException,
    AspectNotOkException;
ContainerRef getCref();
void setCref(ContainerRef cref);
void addAspects(List<IPoint> p, IAspect aspect, Properties aspectProperties);
void removeAspect(IPoint p, IAspect aspect, Properties aspectProperties);
void addCoordinator(Class<? extends Selector> s, ICoordinator c);
List<ICoordinator> getCoordinators();
```

Listing 3: ITransactionLayer interface

This layer is responsible for the transaction management and the execution of the aspects. When a method is invoked, the transaction manager first checks if the transaction is not `null` and that if it is valid (see Section 5.1). If the transaction is null, an *implicit* transaction is created, if it is invalid an exception is thrown. This is realised with the method shown in Listing 4. The method uses the `TransactionManager`, which is explained in Section 2.2.6, for checking and creating the transaction. If a bulk write operation is issued this layer splits the bulk operation into multiple normal write operations. This is necessary in order to simplify the development of the underlying ContainerEngine and the Coordinators.

```
private Transaction initTransaction(Transaction tx) throws InvalidTransactionException {
  if (tx == null) {
    tx = TransactionManager.getInstance().createImplicitTransaction(
        ICapi.INFINITE_TIMEOUT);
  }
  if (!TransactionManager.getInstance().isValid(tx)) {
    throw new InvalidTransactionException("Unknown Transaction: " + tx);
  }
  tx.addContainerRef(this.getCref());
  return tx;
}
```

Listing 4: initTransaction method

After the transaction has been checked the `TransactionLayer` executes the PRE-aspects which are registered on the container. The aspect management is done in this layer because of the following reasons:

- The must aspects only should be executed if the transaction is valid.
- It must be guaranteed that the transaction which is used is valid. Therefore, the aspect does not have to check the passed transaction if it uses it.
- If an aspect answers with NOTOK the transaction has to be rolled back. In this layer every operation has a valid transaction and therefore it can be rolled back.

The management of global and local aspects is explained in Section 2.2.5 in further detail.

Container Engine In this layer the management of the coordinators is done. The interface (shown in Listing 5) is very similar to the `ITransactionLayer`. The methods for aspect management are missing since this is handled in the transaction layer and the write method accepts only one entry instead of a list of entries. This is because of the transaction management which is explained in Section 2.2.5 and to simplify the container engine and the coordinators. A detailed description of the API and all parameters is given in table 10 in Appendix A and the exceptions are explained in Section 4.

```
1  void commit(Transaction txn) throws TransactionLockException;
2  void commitSubTransaction(Transaction txn) throws TransactionLockException;
3  void rollback(Transaction txn) throws TransactionLockException;
4  int getSize();
5  List<Entry> read(Transaction tx, List<Selector> selectors) throws
       TransactionLockException, NoSuchCoordinationTypeException, CountNotMetException;
6  void write(Entry entry, Transaction tx) throws ContainerFullExcpetion,
       TransactionLockException, NoSuchCoordinationTypeException;
7  ICoordinator getCoordTypefromSelector(Class<? extends Selector> s) throws
       NoSuchCoordinationTypeException;
8  List<Entry> shift(Entry entry, Transaction tx) throws TransactionLockException,
       NoSuchCoordinationTypeException;
9  List<Entry> take(Transaction tx, List<Selector> selectors) throws
       TransactionLockException, NoSuchCoordinationTypeException, CountNotMetException;
10 ContainerRef getCref();
11 void setCref(ContainerRef cref);
12 void addCoordinator(Class<? extends Selector> s, ICoordinator c);
13 List<ICoordinator> getCoordinators();
14 int currentSize();
```

Listing 5: IContainerEngine interface

The container engine stores the coordinators in two Java maps, one for the implicit and one for the explicit coordinators. The key of the Map is the corresponding selector class for the coordinator. Since MozartSpaces supports multiple selectors for one operation the container engine has to coordinate the execution of the operation between the

coordinators. In the following the execution of each API operation is described and how it is implemented.

write: first, the write operation is issued on all explicit coordinators for which a selector has been provided, selectors which belong to an implicit coordinator are ignored. Afterwards, the write operation is executed on all remaining implicit coordinators regardless if a selector has been provided or not.

read: the read operation iterates over the selectors and invokes the read operation on the corresponding coordinator. Each coordinator gets the result of the previous selector as argument. This is necessary because in XVSM multiple selectors are evaluated using the Boolean AND operation. The pseudo code in listing 7 illustrates the used algorithm.

```
1 List entries = new List();
2 for(Selector s : selectors){
3   entries = getCoordinator(s).read(s, entries);
4 }
```

Listing 6: pseudo code of the read operation in the container engine

take: this operation uses `read` to determine which entries have to be deleted. Afterwards, it issues the delete operation on all coordinators which store a reference on the entries which shall be deleted.

destroy: in order to log which entries have been destroyed, the operation is mapped to `take` in the blocking layer.

shift: first, the `shift` operation is issued on the explicit coordinators for which a selector has been provided. It is possible that an explicit coordinator can not decide which entry it has to removed (for instance, a key coordinator can shift an entry only if the key which shall be used to store the new entry is already used). In this case, the coordinator throws an exception and the `ContainerEngine` remembers that the shift operation could not remove an entry with this coordinator. Whether or not a coordinator could not remove an entry, the `shift` operation is issued on all implicit coordinators. Finally, the explicit coordinators are executed again which failed at the first attempt because the successful shift on the explicit coordinator removed an entry . Therefore, it is no longer necessary for the implicit coordinator to decide which entry has to be removed (see Section 6.1) This algorithm is illustrated in Listing 7.

```
1 List failedCoordinators = new List();
2 for(Selector s : selectors){
3   Coordinator c = getCoordinator(s);
4   try{
```

```
5     if(c.is explicit()){
6       c.shift(e, s); // e is the entry
7     }
8   }catch(Exception e){
9     failed.add(s); // the selector which failed is stored
10  }
11  for(Coordinator c : implicitCoordinator()){
12    c.shift(e);
13  }
14
15  for(Selector s : selectors){
16    Coordinator c = getCoordinator(s);
17    c.shift(e,s);
18  }
19 }
```

Listing 7: pseudo code of the shift operation in container engine

Coordinators This part of the container is responsible for storing meta information about entries in order to realise coordination models. Coordinators can be created and extended by users and can be added during the creation of a container. Any number of coordinators can be added to a container additionally. But, every container has at least a random coordinator which is added automatically. There can be only one instance of the same coordinator on a container because it is identified by its selector. If more than one instance of coordinator would be added the container engine can not differentiate them. A detailed description of coordinators is given in Section 6.

In Figure 6 the execution of an operation within a container is summarised. When the `execute` method is invoked on the BlockingLayer it evaluates the passed OperationTask and issues the corresponding operation on the TransactionLayer. The TransactionLayer first checks if the transaction is valid and creates an implicit transaction if necessary. Afterwards all PRE aspects are executed and the corresponding method in the ContainerEngine is invoked. The ContainerEngine executes the method based upon the provided selectors on the coordinators. The result is then returned to the ContainerEngine and the TransationLayer. In the TransactionLayer the POST-aspects are executed before the result is returned to the BlockingLayer. The BlockingLayer finally wakes up blocking operations.

2.2.4 Container Management

In MozartSpaces the containers are managed by a class called `ContainerManager`. As shown in Listing 8 the ContainerManager class provides methods to create, delete and get containers. Since all container operations can be executed under a transaction, all operations, except those for meta- (`getMetaContainer()`, `getContainer(ContainerRef metaCref)`) and container-container (`getContainerContainer()`) (this container stores

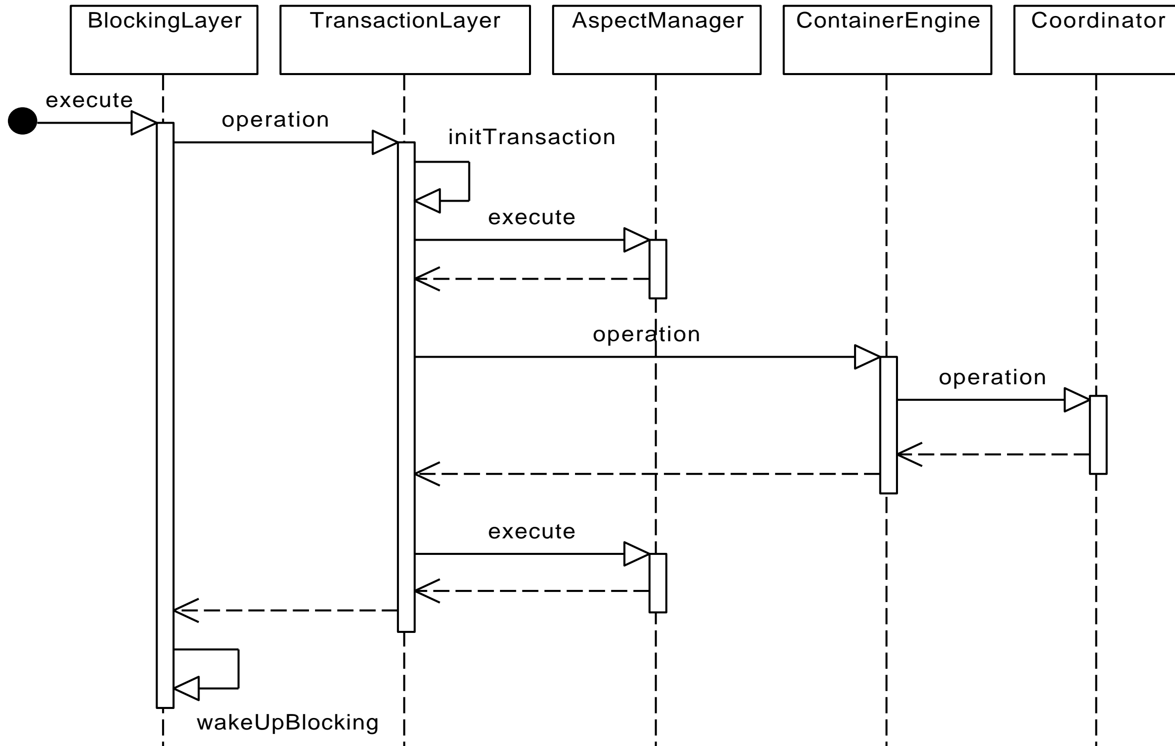

Figure 6: Sequence diagram illustrating the successful execution of an operation within a container.

the container as described below) handling, accept a transaction parameter. The main purpose of this class is to manage the creation of containers, storing the instances of containers (see Section 2.2.3), make them accessible using the container reference or name and handle the destruction of containers. Additionally, the container manager creates a `TimeoutHandler` [41] for each container which takes care of the timeouts of blocking operations within the container. A detailed description of all methods and the parameters is given Table 11 in Appendix A.

```
1 ContainerRef createContainer(Transaction tx, String name, int size);
2 ContainerRef createContainer(Transaction tx, String name, int size, ICoordinator...
      coordinators);
3 void deleteContainer(Transaction tx, ContainerRef cref);
4 IContainer getContainer(Transaction tx, ContainerRef cref);
5 ContainerRef getContainer(Transaction tx, String name);
6 IContainer getContainer(ContainerRef metaCref);
7 ContainerRef getMetaContainer(Transaction tx, ContainerRef cref);
8 List<ContainerRef> getAllContainer(Transaction tx);
9 IContainer getContainerContainer();
```

Listing 8: Interface of the Container Manager

In MozartSpaces the ContainerManager uses an XVSM container to store the containers. For this purpose, it creates a container which has a key- and a linda-coordinator. Since internally it is possible to store every Java object in an atomic entry the container

manager uses the following tuple to store the container references:
`[ContainerRef cref, IContainer container, ContainerRef metaCref]`
The `cref` is the container reference of the container, `container` is a Java reference to the container instance and `metaCref` is the container reference of the meta container. In the tuple which is written for a meta container the `cref` and the `metaCref` are the same because a meta container does not have another meta container. If a named container is created the key coordinator is used to store the name of the container. In this case the name of the container is used as key.

In order to speedup the look-up process of a container the class, which internally represents a container reference (`ContainerRef`), has a reference to the container to which it belongs. Therefore the look-up of a container is only performed at the first time the container reference object is used. Afterwards the direct reference is used.

2.2.5 Execution of Aspects

Since MozartSpaces supports local (on operations on a single container) and global (on operations on the run-time) aspects there are two points where aspects are handled: a local aspect manager in the transaction layer of each container (see Section 2.2.3) which handles the local aspects and a global aspect manager which handles the global aspects. For this purpose two classes are used: `GloabalAspectManager` and `AspectManager`. The global aspect manager is a wrapper class over the aspect manager which takes care of thread synchronisation which is done in the blocking layer in case of local aspects. The local aspects are triggered by the transaction layer in the container. Global aspects are triggered by the tasks (see Section 2.2.2) which are used for executing the operation. A detailed description of the aspects in MozartSpaces is given in [41].

2.2.6 Transaction Management

The transaction management in MozartSpaces is done at two points, the transaction layer in the container (as described in 2.2.3) and in a class called `TransactionManager`. The transaction manager handles the creation of transactions and it stores a list of active transactions. This list is used to check if a transaction, which is used by a client, is valid within the core. Since MozartSpaces does not support distributed transactions, transactions can only be used at the core instance on which they were created. The implementation of the transaction management is explained in Section 5 in more detail.

2.2.7 Timeout Management

The management of timeouts is realised in MozartSpaces by using the Java `Scheduled-ThreadPoolExecutor` (see Section 2.2.1) thread pool implementation. Basically, there

are two kinds of timeouts which have to be checked, timeouts on operations (defining the lifetime of an operation) and timeouts on transaction (defining the lifetime of a transaction). The first are checked for each container separately. Whenever a new container is created, an instance of `TimeoutHandler` is created and passed to the scheduled thread pool. `TimeoutHandler` is an inner class of the `ContainerManager` which just invokes the `updateTimeouts()` (see Section 2.2.3) method for its container. The checking of the operations is then done within the container implementation. For this purpose the `BlockingLayer` of the container iterates over all blocking operations (which are represented by `Task` classes as described in 2.2.2). The `Task` class provides two methods for timeout handling:

`updateTimeout()` updates the private variables which are used for timeout handling. For each `Task` the time it has been created, the time when the timeout has been updated at last and the remaining timeout are stored. This method calculates the remaining timeout by using the code shown in Listing 9.

```
1  public void updateTimeout() {
2    if (this.timeout != ICapi.INFINITE_TIMEOUT) {
3      long now = System.currentTimeMillis();
4      long timepassed = now - this.lastTimeoutUpdate;
5      this.timeout = timeout - timepassed;
6      if (this.timeout == ICapi.INFINITE_TIMEOUT) {
7        this.timeout = 0;
8      }
9      this.lastTimeoutUpdate = now;
10   }
11 }
```

Listing 9: method which is used to calculate the remaining timeout

First, it is checked whether the timeout has to be updated by checking if the timeout is set to the constant `INFINITE_TIMEOUT` which is defined in the `ICapi` interface[7]. If the timeout has to be updated, the method calculates the remaining timeout by deducting the time which passed from the last update from the timeout which has been calculated by the last invocation of this method. It could happen that the now calculated remaining timeout has the same value as `INFINITE_TIMEOUT` constant, therefore this is checked and if so the timeout is set to zero.

`timeoutExpired()` checks if the timeout of the task is expired. If so, the method returns `true` otherwise `false`. This is realised by checking if the timeout is smaller or equals to zero.

If the blocking layer finds `Task`s where the timeout has expired it sets the result of the task to an exception and removes it from the blocking operations. An exception will then

[7]in the current implementation INFINITE_TIMEOUT is set to `-1`

be sent to the client informing it that the operation could not be fulfilled within the given timeout.

The timeout handling of transactions works equally to that of the blocking operations. A `TimeoutHandler`, which is an inner class of the `TransactionManager` periodically[8] checks all active transactions if the timeout has expired (using the same algorithm). If so, the transaction is rolled back. The client gets informed about this as soon as it invokes another operation using this transaction (including commit and rollback) because the core will inform the client that the transaction is invalid.

Details of timeout handling and the semantics of the timeouts in MozartSpaces are covered in [41].

2.2.8 Remote Communication

Another part of the core which can be extended or replaced by an user is the remote communication layer. A user can configure which transport technology shall be supported by a MozartSpaces instance and which encoding protocol for messages shall be used for the communication. Currently, the only supported transport technology is sending and receiving of messages over a TCP connection and MozartSpaces instances can communicate using serialised Java objects or the standardised XML protocol which is described in Section 7. The combination of transport and encoding protocol is determined by the MozartSpaces URL scheme. For instance, if the URL scheme uses `tcpxml` as protocol (e.g., `tcpxml://localhost:9876`), XML messages will be sent over a TCP connection.

The classes involved in the communication are depicted in the UML class diagram in Figure 7. The main class which controls the communication is the `TransportHandler`. This class is used as a gateway between the communication layer and the core. If the core has to send a message it uses the method `send` which accepts a task as parameter and if a message has been received `processTask` is used to pass the received message to the core. The TransportHandler is implemented as a singleton, therefore the `getInstance` method has to be used to get a reference to the TransportHandler. The `init` method can be used to reconfigure the instance. In this method the MozartSpaces configuration file (see Section 3) is used to determine which transport technologies and which message types are supported and how they have to be configured. Based upon the information from the configuration file the TransportManager initialises the communication layer. Additional methods exist to cleanly shutdown a TransportHandler which frees all resources and to obtain the available listener and sender.

Every transport protocol has to support the listening part which waits for incoming messages and the sending part which is used to send messages. The listening part of a communication has to implement the `ITransportListener` interface and the sending

[8]The frequency can be configured as described in Section 3

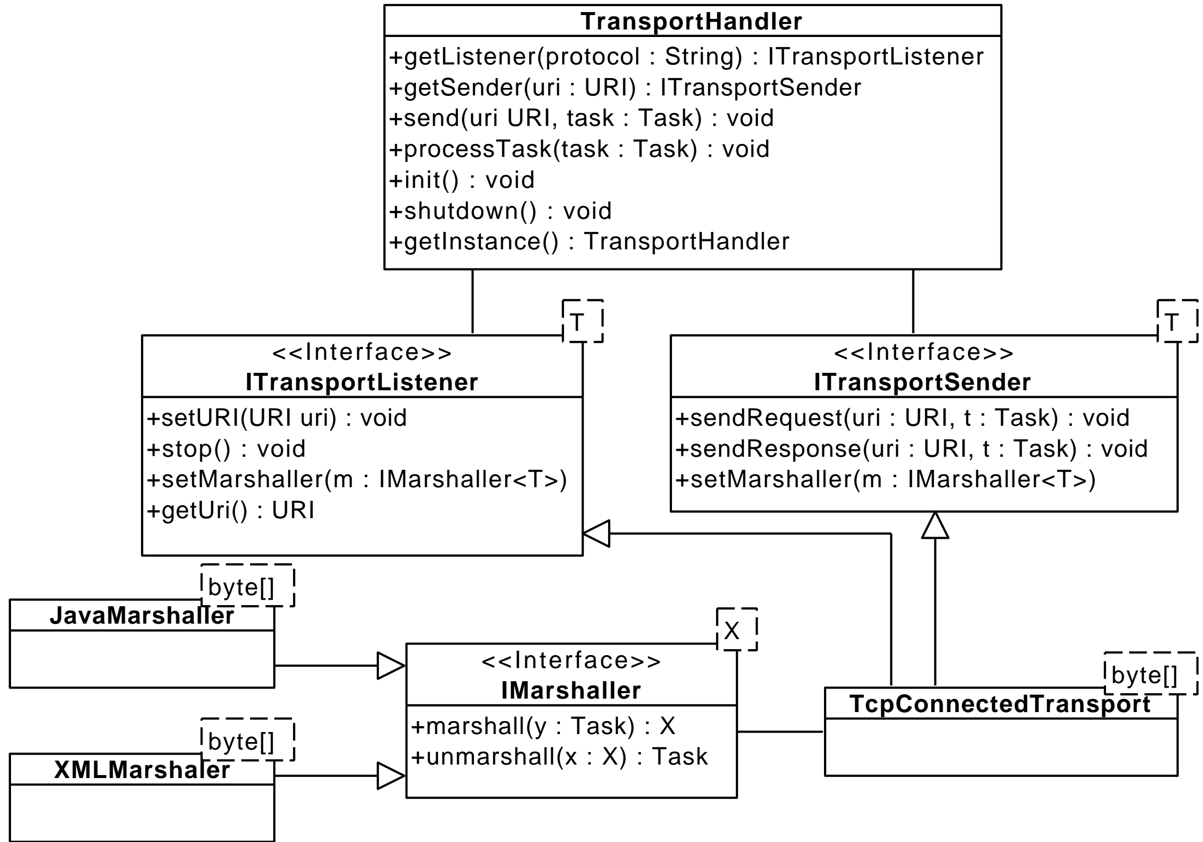

Figure 7: Class diagram of the classes involved in the remote communication.

part `ITransportSender`. An implementation of ITransportListener has to start listening for new messages after the initialisation. The method `stop` is invoked when the listener shall be stopped. Additionally, getter and setter methods for the URI on which the listener is listening are defined and a method `setMarshaller` which is used to inject the marshaller (see below). The ITransportSender interface defines methods for sending a request (`sendRequest`) and a response (`sendResponse`) to a remote core. The target of the message can be determined from the container reference of the answer container (which is stored within the task). A method for setting the marshaller (`setMarshaller`) is also defined.

The `IMarshaller` interface has to be implemented by the classes which translate the internal task objects into the messages format which shall be transmitted to the remote core instances. The objects which do the transformation are called marshallers. IMarshaller defines two methods, one for marshalling task into the format which shall be sent and one to unmarshall received messages. The marshaller is injected into the sender and the receiver implementation during the initialisation.

All three interfaces are generic. The type parameter defines the type of message which is sent. This is necessary because some transport mechanisms might require a special type of message. As mentioned above, currently a tcp transport is provided which is

implemented in `TcpConnectedTransport`. This transport implements both, the ITransportListener and the ITransportSender interfaces. The tcp transports sends and receive byte arrays, therefore the type parameter is set to `byte[]`. Currently, two implementations of IMarshaller exist, the `JavaMarshaller` which serialises the Java objects into byte arrays and `XMLMarshaller` (see Section 7) which transforms the Java objects into XML messages. Both initialise the type parameter to `byte[]` because they are used with the tcp transport which only supports this data type. In order to use the XML marshaller with a transport which requires strings (for example Jabbber [4]) a simple wrapper class can be used. The byte arrays used by the current marshaller implementation can be transformed to strings by using: `new String(theByteArray)` and the strings can be transformed to byte arrays by using: `theXMLString.getBytes()`.

The current implementation of the transport protocol, the `TcpConnectedTransport`, sends and receives messages over a TCP connection. For optimisation reasons, the connection is not teared down after each message. It is very likely that during the communication between two XVSM instances multiple messages are exchanged. Therefore, the connection between two peers remains open after a message has been sent and will be reused. This reduces the overhead which is caused by the creation of the connection. The `TcpConnectedTransport` internally uses a Java `ExecutorService` to process the requests. For each new incoming message, a `Runnable` is created and queued in the executor service. This has been done to avoid the creation of a new thread for each request and to better fit the run-time architecture (see Section 2.1). The implementation of `Runnable` is called `TcpListenerThread`. This is an inner class of the `TcpConnectedTransport`. When the `TcpListenerThread` is executed, it first reads the bytes received in an byte array. Afterwards it uses the marshaller to unmarshal the received message into a `Task`. Finally, the thread passes the received task to the `TransportHandler` to execute the operation within the core.

3 Dynamic Kernel Configuration

MozartSpaces provides a dynamic kernel configuration mechanism. The configuration parameter can be specified in a configuration file called `spaces.prop` per default which is in the Java properties file format[9].

The configuration of MozartSpaces is handled by a class called `ConfigurationManager`. The interface of this class is depicted in listing 10. The implementation is realised using the singleton design pattern [27], therefore methods for getting and removing the instance are provided. The `init` method has to be used to initialise the ConfigurationManager. During initialisation, the ConfigurationManager checks if already a configuration file is available (in the working directory of MozartSpaces), if so, it uses the existing one, if not, it creates a new one and sets the parameters to default values (see Section 3.2). The name of the configuration file is per default `spaces.prop` but this and the location of the file can be changed in the constructor of the ConfigurationManager. Besides the methods to control the life cycle of the configuration manager, methods for getting and setting parameters are provided. A detailed description of all methods and parameters of the interface is given in Table 12 in Appendix A.

```
1 static ConfigurationManager getInstance();
2 static void removeInstance();
3 static void init(String configurationFile) throws IOException;
4 synchronized boolean setStringSetting(String key, String value);
5 String getStringSetting(String key);
6 int getIntegerSetting(String key);
7 long getLongSetting(String key);
8 boolean getBooleanSetting(String key);
```

Listing 10: ConfigurationManager interface

3.1 Reconfigure during Run-time

The `ConfigurationManager` class can be used to reconfigure a MozartSpaces instance during run-time. An user can get a reference to the `ConfigurationManager` by issuing `ConfigurationManager.getInstance()`. Afterwards, the `setStringSetting` method can be used to set the configuration parameter. Depending on which parameters have been modified the parts of the MozartSpaces core which are affected by these parameters have to be reinitialised. If the configuration of the remote layer is modified the `TransportHandler` class has to be reinitialised. This can be done by issuing `TransportHandler.getInstance().init();`. If the thread pool configurations have been modified the thread pool wrapper classes have to be reinitialised by issuing `TimeoutSchedulerPool.init();` and `EventProcessingPool.init();`.

[9] `http://java.sun.com/j2se/1.5.0/docs/api/java/util/Properties.html`

Alternatively, the `ConfigurationManager` instance could be destroyed (`removeInstance()`) and reinitialised with a modified configuration file. Afterwards, the parts of the core which are affected by the changes have to be reinitialised as described above.

3.2 Parameters

In this section the parameters which can be configured are explained. A description of all parameters and their default parameters is given in the following list.

EventProcessingThreads=100 :
The maximum number of threads in the main thread pool. The default value is `100`.

SchedulerMaxThreads=1 :
The maximum number of threads within the scheduler thread pool. The default value is `1`.

SchedulerTimeoutDelay=500 :
The delay of the threads within the scheduler thread pool in milliseconds. The default value is `500` ms. If this value is too high, the detection of timed out operations would last very long. Therefore, an operation could be fulfilled even when the timeout already expired. On the other hand, if the value is too low the task would be scheduled very often which could lead to a live lock in the worst case.

ReplySenderThreads=100 :
The number of threads in the thread pool of the TCP receiver and sender component. The default value is `100`.

EnableRemoteAccess=true :
Indicates if the core shall be accessible over the network. If this value is `true` the following configuration parameters will be processed otherwise they will be ignored because they are not relevant. The default value is `true`.

RemoteProtocols=TcpJava|TcpXML :
Contains a list of transport mechanisms (transport and encoding protocol) which are supported by the core instance. The list items have to be separated with "|". Currently TcpJava and TcpXML is supported. The strings which are used here will be used to identify the additional configuration parameter of the transport. Every transport mechanism has to provide the class of the sender and the listener implementation, the URI on which the listener shall be listening and the marshaller which shall be used. If additional parameters are necessary for a transport technology they should be added with the identification of the transport as prefix.

TcpXML.sender=org.xvsm.remote.tcpconnection.TcpConnectedTransport :
The implementation of the `ITransportSender` interface of the TcpXML transport.

TcpXML.uri=tcpxml://localhost:9876 :
The URI on which the TcpXML transport is listening. Per default it is listening on port `0`, which means that a random free port will be used. This is necessary to enable multiple instances of MozartSpaces on one machine without manually configuring each one. In this example `9876` is used as port. The uri has to be unique, e.g., there can not be two transports with the same URI.

TcpXML.listener=org.xvsm.remote.tcpconnection.TcpConnectedTransport :
The implementation of the `ITransportSender` interface of the TcpXML transport.

TcpXML.marshaller=org.xvsm.remote.marshaller.XMLMarshaller :
The implementation of the `IMarshaller` interface of the TcpXML transport.

TcpJava.listener=org.xvsm.remote.tcpconnection.TcpConnectedTransport :
The same as TcpXML.listener.

TcpJava.sender=org.xvsm.remote.tcpconnection.TcpConnectedTransport :
The same as TcpXML.sender.

TcpJava.uri=tcpjava://localhost:4321 :
The same as TcpXML.uri. The default port is as well `0`. In this example it is `4321`.

TcpJava.marshaller=org.xvsm.remote.marshaller.JavaMarshaller :
In this configuration parameter the two default transports differ (besides the uri). `TcpXML` uses the `XMLMarshaller` and `TcpJava` used the `JavaMarshaller`.

DefaultAnswerToProtocol=TcpJava :
This is the protocol which shall be used as default protocol of the answer container. The default value is `TcpJava`.

In addition to the above MozartSpaces configuration parameters the logging system of MozartSpaces can be configured. Currently, *log4j*[10] is used. The position of the log4j configuration file can be configured using the `log4j.configuration` system parameter. A default log4j configuration file is provided in the MozartSpaces class path and will be used if log4j.configuration is not set. The format of this property file can be found in the log4j documentation.

[10] `http://logging.apache.org/log4j`

4 Exceptions

In this section the exception handling within the MozartSpaces core is described. First, an overview of the internal handling of exceptions is given. Afterwards, all exceptions which can occur during the execution of operations are explained.

4.1 Exception Handling

In MozartSpaces Java exceptions are used for the communication between internal components and to inform components of erroneous situations. The super class of all exceptions, except `FatalException`, thrown in MozartSpaces is `XCoreException` which extends the Java class `Exception`. The `FatalException` extends the Java `RuntimeException`[11] and indicates an exception within the core which was not expected. Normally, the occurrence of this exception is an indication for a bug within MozartSpaces. The inheritance hierarchy of the exceptions is depicted in Figure 8. A detailed description of all MozartSpaces exceptions is given in Section 4.2.

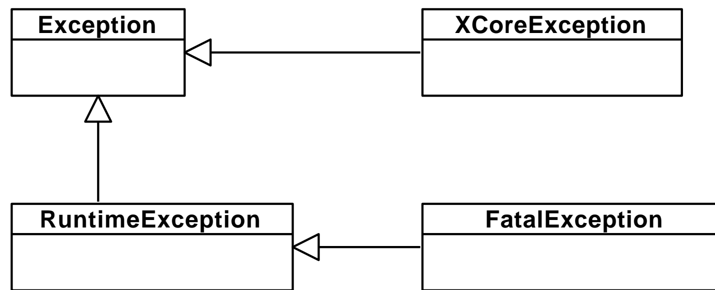

Figure 8: Inheritance hierarchy of the MozartSpaces exceptions.

Since MozartSpaces executes all operations asynchronously (see Section 2.2.2), the exceptions can not be forwarded to the calling methods by using the stack trace in some cases. Instead, the exception is set as result within the `Task` object which represents the operation. The component which forwards the result of an operation to the calling client has to check whether an exception occurred or not (if the core is running embedded the `Capi` class handles this, if it is remote, it is done by the marshaller implementation). This check can be performed by checking if the result object is an instance of the Java class `Throwable`. If so, the Capi can re-throw the exception and inform the client about the erroneous state. In order to simplify this process, all methods of the `ICapi` interface only declare to throw `XCoreException`. Therefore, the ICapi implementation does not

[11] In contrast to normal exceptions a `RuntimeException has not to be declared in the signature of a Java method (see http://java.sun.com/j2se/1.5.0/docs/api/java/lang/RuntimeException.html).`

have to check which exception has to be thrown, this check can be done by the client if necessary.

4.2 Exception Description

In this section all exceptions which can occur during the execution of operations are explained.

AspectNotOkException This exception can be used by aspect implementers to indicate that an aspect's result is not OK and therefore the execution of the operation shall be stopped and the transaction shall be rolled back (see Section 1.1.3).

AspectRescheduleException This exception can be used by aspect implementers to indicate that the operation shall be rescheduled. I.e., the operation shall be stopped and re-added to the thread pool for later execution (see Section 1.1.3).

AspectSkipException This exception can be used by aspect implementers to indicate that the operation shall be skipped. I.e., the operation shall not be executed on the container and the remaining PRE aspects, instead, the post aspects shall be executed immediately. This exception is only allowed for PRE aspects (see Section 1.1.3).

CannotShiftException This exception can be used by implementers of a custom coordinator to indicate that the coordinator can not decide which entry shall be shifted (see Section 6).

ContainerFullException This exception indicates that the container has not enough free place for entries which shall be written. This exception is thrown in the `ContainerEngine` (see Section 2.2.3) but it can also be used by coordinator implementers to indicate that the entry can not be written into the coordinator (see Section 6).

ContainerLockedException This exception is thrown when an operation is executed on a container but the container is already locked by another transaction (see Section 5).

ContainerNameOccupiedException This exception is thrown by the `ContainerManager` (see Section 2.2.3) if a named container shall be created but the name is already used by another container.

CountNotMetException This exception is thrown while execution of an operation within a coordinator if the required amount of entries is not available.

EntryLockedException This exception is thrown if an operation is executed on an entry but the entry is already used by another transaction. It can also be used by coordinator implementers. In the `ContainerEngine` the exception is mapped to a `TransactionLockedException` (see Section 5).

FatalException This exception indicates an invalid state within the kernel. Every unexpected exception which occurs is mapped into a `FatalException`. Therefore it is normally an indication for a bug in MozartSpaces. The exception which caused the `FatalException` is set as cause can be retrieved with `fatalException.getCause();`

InvalidContainerException This exception is thrown by the `ContainerManager` if an unknown container reference is used. This can happen if a user uses a container reference which was not been created with `createContainer` on the MozartSpaces instance or the container has been deleted in the meantime.

InvalidTransactionException This exception is thrown by the `TransactionManager` if a transaction is used by the client which is not valid. For example, the transaction has already be committed or rolled back (see Section 5).

NoSuchCoordinationTypeException This exception is thrown by the `ContainerEngine` if a selector is used which does not have a corresponding coordinator on the container (see Section 2.2.3).

TimeoutExpiredException This exception is thrown if a blocking operation times out.

TransactionLockException This exception is thrown if an entry is used by another transaction during the execution of an operation. It can be used by a coordinator implementer to indicate this and is caught in the blocking layer (see Section 2.2.3).

XCoreException This is the abstract super class of all MozartSpaces exceptions (except FatalException).

XCoreRemoteException This exception can be thrown, or extended if necessary, by implementers of transport technologies (see Section 2.2.8).

4.3 ICapi Exceptions

As described above all methods declared in `ICapi` interface declare to throw `XCoreException`. In Table 6 an overview of the methods is given and which exception may be thrown. A detailed description of the `ICapi` interface (the methods and all parameters) is given in [41].

Method	Exceptions
`addAspect`	`XCoreRemoteException`
`clearSpace`	`XCoreRemoteException`
`commitTransaction`	`XCoreRemoteException, InvalidTransactionException`
`createContainer`	`XCoreRemoteException, InvalidTransactionException, ContainerNameOccupiedException`
`createNotification`	`XCoreRemoteException, InvalidContainerException`
`createTransaction`	`XCoreRemoteException`
`destroy`	`XCoreRemoteException, TimeoutExpiredException, InvalidTransactionException, InvalidContainerException, CountNotMetException, NoSuchCoordinationTypeException`
`destroyContainer`	`XCoreRemoteException, InvalidTransactionException, InvalidContainerException`
`lookupContainer`	`XCoreRemoteException, InvalidTransactionException`
`lookupMetaContainer`	`XCoreRemoteException, InvalidTransactionException, InvalidContainerException`
`read`	`XCoreRemoteException, TimeoutExpiredException, InvalidTransactionException, InvalidContainerException, CountNotMetException, NoSuchCoordinationTypeException`
`removeAspect`	`XCoreRemoteException, InvalidContainerException`
`rollbackTransaction`	`XCoreRemoteException, InvalidTransactionException`
`shift`	`XCoreRemoteException, InvalidTransactionException, InvalidContainerException`
`shutdown`	`XCoreRemoteException`
`take`	`XCoreRemoteException, InvalidTransactionException, InvalidContainerException, CountNotMetException, NoSuchCoordinationTypeException`
`write`	`XCoreRemoteException, TimeoutExpiredException, InvalidTransactionException, InvalidContainerException, ContainerFullException`

Table 6: Possible exception of the ICapi interface

5 Transactions

This section describes the implementation of the transaction mechanism in MozartSpaces. The space offers pessimistic transaction control providing ACID properties. ACID has been defined for database systems and stands for [30]:

Atomicity. All operation which belong to a transaction shall be seen as one unit which can not be split. If one operation of the transaction is established all operations have to be established.

Consistency. After the finalisation of a transaction the database has to be in a consistent state (i.e., it has to be in a state which is defined by the database scheme). During the execution of the transaction the consistency may be broken, but at the end it has to be consistent otherwise the transaction has to be reversed.

Isolation. This means that concurrent transactions must not affect each other. Uncommitted changes of transactions must not be visible to other components.

Durability. The changes which have been done during a transaction have to be permanent after the transaction has been committed. This property is interesting for DBS and spaces which offer persistency. In MozartSpaces persistency will be supported in one of the next versions.

5.1 Transaction Management

In MozartSpaces every transaction is represented by a `Transaction` object. Every transaction has a identifier which is unique within a core instance. Besides the identifier it stores the following information:

- a transaction stores information for the timeout handling (see Section 2.2.7)
- a list of coordinators which where used during the execution of the transaction
- a list of containers which where used during the execution of the transaction
- a map of entries which where modified during the execution of the transaction. The key of the map is the container reference in which the entry is stored.
- the URI of the MozartSpaces instance which was used for creating the transaction (this is used by commit and rollback to send the operation to the correct MozartSpaces instance)
- the transaction which is the father of this transaction (see Section 5.2). If the transaction does not have a father the field is `null`

- a field indicating if the transaction is implicit or explicit
- a field indicating if the transaction is currently rolling back

The life-cycle of a transaction is controlled by a class called `TransactionManager`. The interface of the transaction manager is depicted in Listing 11. It provides methods for creating transactions, subtransactions and implicit transactions (see Listing 11). Implicit and explicit transactions can have a timeout which has to be passed during the creation. If the timeout shall be infinite, the constant `ICapi.INFINTE` has to be passed[12]. A sub-transaction (see Section 5.2) inherits the timeout from the father transaction, therefore no timeout parameter is necessary. That is because sub-transactions can not live longer than their father transaction. A method is provided to get all currently active transactions. This method is used by the `ClearTask` (see Section 2.2.2) to rollback all transactions when cleaning the space. The method to get the transaction reference is used by the `OperationTask` (see Section 2.2.2). This is necessary because only the transaction identifier is sent to a (remote) client but for the internal processing the transaction object is necessary. The method isValid can be used to determine if a transaction is valid. A transaction is valid if it has been created at this MozartSpaces instance and has not been committed or rolled back yet, otherwise it is invalid. The transaction manager also controls the execution of a commit and a rollback which can be initialised with the corresponding methods. Finally, an operation is provided to remove a transaction from the transaction manager. This method is used by the `TransactionTask` after a transaction has been committed or rolled back.

```
1 void commitTransaction(Transaction tx)
2 Transaction createImplicitTransaction(long timeout)
3 Transaction createSubTransaction(Transaction father)
4 Transaction createTransaction(long timeout)
5 List<Transaction> getAll()
6 Transaction getTransaction(String id)
7 booelan isValid(Transaction tx)
8 void removeTransaction(Transaction tx)
9 void rollbackTransaction(Transaction tx)
```

Listing 11: Interface of the TransactionManager

5.2 Sub-transactions

Internally MozartSpaces supports sub-transaction. A sub-transaction is a part of the enclosing (or father) transaction, but if the sub-transaction is rolled back, the father transaction can continue to execute. If a sub-transaction is committed all locks (see next sections) are transferred to the father transaction, therefore the changes of a committed sub-transaction can be undone by rolling back the father transaction. Since sub-transactions

[12]The constant `ICapi.INFINTE` is currently set to `-1`

are part of the father transaction operations executed under the sub-transaction can see all modifications of the father transaction even if it has not been committed. Support for sub-transactions has been added to MozartSpaces for two reasons:

- When an operation has to block because it can not be fulfilled at the moment, all side effects, which have already been done by this particular operation, have to be undone but the transaction, which was used by the client to execute the operation, shall stay valid. Therefore, each attempt to execute an operation is done within a new sub-transaction of the transaction used by the client. If the operation has to be blocked, the sub-transaction is rolled back. If the operation succeeded, the sub-transaction is committed and all locks are transferred to the father transaction. Using this sub-transaction mechanism allows to reuse the transaction mechanism for the blocking operation problem without the need to implement additional logic.
- As described in Section 2.2.3 bulk write operations are split into single write operations by the `TransactionLayer`. Since the bulk operation shall be atomic the single write operations are executed using sub-transactions. If one of the write operations does not succeed within the timeout, the sub-transaction is rolled back. The transaction used (by the client) to execute the bulk operation is not modified if the sub-transaction fails but the modifications of the sub-transaction are inherited by the father transaction on commit which makes them appear as if they where done by the father transaction. Again, this approach allows the usage of the transaction mechanism for a problem which otherwise would need a special implementation.

5.3 Locking

Since MozartSpaces uses pessimistic transactions a locking mechanism is used to prevent modifications of one object concurrently by two transactions. For this purpose a fine grained locking mechanism is used. During the execution of an operation entries and containers can be created, deleted or modified. Therefore, these two objects have to be protected from other transactions to ensure isolation. In MozartSpaces the mutual exclusion can be done for a whole container or for each entry separately. The implementer of a coordinator can choose whether to support concurrent transactions for a whole container or for the entries. The second solution offers a much higher performance because only entries are locked which really have been used under the transaction and should be preferred if possible. The first solution, the locking of the whole container, should only be used if entry locking is not possible because it allows only one active transaction to access the container concurrently. All operations using another transaction have to wait until the transaction has terminated. The granularity of the locking is defined by the coordinator. Locks are represented by fields within the objects (the coordinator or the

entry). The type of the field is `Transaction` and is set to the transaction which currently holds the lock on the object. If the field is set to `null` no lock is set.

Regardless if container or entry locking is used the locks of the entries will be set correctly. This happens in the `ContainerEngine` therefore the coordinator implementer does not have to take care of setting the locks on entries. The `ContainerEngine` sets the locks on the result (the entries returned by the selection) of the coordinator after an operation has been executed successfully. This approach has been chosen because in some circumstances one coordinator alone can not decide whether an entry has to be locked or not. For example, if a read operation is performed with several selectors. then the result of the first coordinator is passed to the second coordinator and so on. The result of the selection is the result of the last selector. If the first coordinator would lock all entries which were selected it is possible that a lot of entries are locked which would not be necessary because they are not part of the final result. In the following two sections it is explained how the container and the entry locking are working and how they have to be used.

5.3.1 Container Locking

This is the simplest form of locking within MozartSpaces. Only one transaction can access a container at the same time and therefore the entry locks can be ignored. On the other hand, this locking method has negative effects on the performance of the system when multiple transactions are used concurrently on the container. Since the granularity of the locking is determined by the coordinators, the abstract `ICoordinator` class offers three methods which can be used to handle the locks. In this implementation it is possible that multiple transactions can read from a container concurrently but only one transaction can use the container (reading and writing) if a write operation is executed. Therefore a write lock can only be granted if there are no other read and write locks (except the locks are from the same transaction which requires the write lock).

```
1  protected void acquireLock(boolean write, Transaction tx)
2      throws TransactionLockException {
3    if (this.writeLock != null) {
4      if (!this.writeLock.isAncestorOf(tx)) {
5        throw new TransactionLockException(this.writeLock);
6      }
7    }
8    if (write) {
9      if (this.readLocks.size() != 0) {
10       Transaction readLock;
11       Iterator<Transaction> iter = this.readLocks.values().iterator();
12       while (iter.hasNext()) {
13         readLock = iter.next();
14         if (!readLock.isAncestorOf(tx)) {
15           throw new TransactionLockException(readLock);
16         }
```

```
17        }
18        this.readLocks.remove(tx.getId());
19      }
20      this.writeLock = tx;
21    } else {
22      if (tx != null && tx.getId() != null) {
23        this.readLocks.put(tx.getId(), tx);
24      }
25    }
26  }
```

Listing 12: Implementation of acquireLock method

Listing 12 depicts the method which can be used for acquiring a lock from `ICoordinator`. The code for thread synchronisation, logging and the comments have been removed. The method expects two parameters, the first one indicates if a write or a read lock shall be acquired, the second one is the transaction for which the lock shall be acquired. First, it is checked if a write lock has already been set on this object. If a write lock is already set, and the transaction holding the lock is not an ancestor of the transaction, the new lock cannot be set. Otherwise the lock can not be granted which is indicated by a `TransactionLockException`. The method `isAncestorOf` is defined in the `Transaction` class and checks if the transaction is an ancestor (father, grandfather, ...) of the transaction passed as argument. If no write lock is set, the method checks if a write or a read lock shall be set. In the case of a write lock, it is checked if there are read locks on the entry. If a read lock is set which is not an ancestor the write lock can not be granted. If the read lock is from an ancestor it can be upgraded to a write lock. This is done by removing the read lock and setting the write lock. If a read lock shall be set it is just checked if the transaction is not null and the transaction is added to the read locks. To optimise the removing of a transaction the read locks are stored in a map where the id is the key and the transaction is the value.

```
1  protected void commitLocks(Transaction tx) throws TransactionLockException {
2    if (this.readLocks.get(tx.getId()) != null) {
3      this.readLocks.remove(tx.getId());
4      if (tx.getFather() != null) {
5        this.readLocks.put(tx.getFather().getId(), tx.getFather());
6      }
7    }
8    if (this.writeLock != null && this.writeLock.equals(tx)) {
9      this.writeLock = tx.getFather();
10   }
11 }
```

Listing 13: Implementation of commitLocks method

The code which can be used to commit a transaction is shown in Listing 13. Again, the code for synchronisation, logging and the comments have been removed. This method delegates all locks to the father transaction if a sub-transaction is committed and, to

clears the lock if a "normal" transaction is committed. Since the indicator of a father transaction is set to `null` if there isn't any, the two cases have not to be distinguished for committing a write lock. First, it is checked if the transaction holds a read lock, if so, the read lock is removed and a lock to the father transaction is added. Afterwards, it is checked if the transaction holds a write lock and updates the lock to the father if necessary.

```
1 protected void rollbackLocks(Transaction tx)
2     throws TransactionLockException {
3     this.readLocks.remove(tx.getId());
4     if (this.writeLock != null && this.writeLock.equals(tx)) {
5       this.writeLock = null;
6     }
7   }
```

Listing 14: Implementation of rollbackLocks method

The last method provided for a coordinator developer is `rollbackLocks` which is depicted in Listing 14. This method just has to remove all locks which are set by the transaction. First it removes a potential read lock. Since a map is used to store the read locks it is not necessary to check if a read lock is present. Afterwards, it is checked if the transaction to be rolled back has a write lock on the coordinator. If this is the case the write lock is set to `null`.

These three methods suffice to implement locking for a coordinator. A developer of a coordinator has to invoke the `acquireLock` method before the transaction produces any side effects. Typically, this is done as first operation in the code. If the lock can not be acquired an exception (`TransactionLockException`) is thrown which will be caught in the `BlockingLayer` of the container in order to block the operation. Commit and rollback can be executed by invoking the corresponding methods in the abstract ICoordinator class.

5.3.2 Entry Locking

As described before, container locking can have a negative effect on the performance of the system. Therefore all built in coordinators are using entry locks instead.

Entry locks are stored in the `Entry` object. There are fields for write lock and read locks in the `Entry` object analogously to the `ICoordinator`. Additionally, the entry has a field for a delete lock which is set if the entry has been deleted by a take or destroy operation. This is not necessary if container locking is used because a container which has been deleted cannot be used to execute operations on it further on. Like for container locking, MozartSpaces provides the following methods for the coordinator developer to handle the transaction management. These methods are defined in the `Entry` object.

When it is checked if an transaction is the ancestor of another transaction it is also

checked if the opposite is the case. This has been implemented to handle the case when an faulty coordinator implementation does not correctly propagated the locks of an object to the father transaction after the commit.

`void setDeleteLock(Transaction deleteLock);`

This method can be used to set a delete lock. If already a lock is set which is not from an ancestor of an `EntryLockedException` will be thrown. This exception can be re-thrown in the coordinator if it is not possible to select another object for deletion[13]. It will be mapped to a `TransactionLockedException` in the `ContainerEngine` and re-thrown. This exception is finally caught in the `BlockingLayer` and the operation will block. A delete lock can only be set if all of the following conditions are fulfilled:

- no delete lock has been set by another transaction. Note that if a delete lock exists which terms from an ancestor of the transaction which tries to acquire the lock then the delete lock can be set.
- no write lock is set from another transaction. Note that if the write lock is from an ancestor, the delete lock can be granted because an entry can be written and afterwards deleted in the same transaction.
- no read locks from other transactions are set. Note that if a read lock from an ancestor exists the delete lock can be granted.

`Transaction getDeleteLock();`

This method can be used to get the current delete lock. If no delete lock is set, `null` will be returned.

`void removeDeleteLock();`

This method removes the delete lock from an entry without any checks. It should only be used if there is certainly no delete lock from another transaction set.

`boolean hasDeleteLock(Transaction tx);`

This method checks if the entry has been deleted by the given transaction or by an ancestor of it. If no delete lock is set or if a delete lock is set by another transaction, `false` is returned; otherwise `true`. This method can be used to check whether an entry has been deleted during the execution of the transaction.

[13]Whether another object can be select depends on the semantics of the coordinator. For example, a destroy with a random coordinator could catch the exception and select another entry for deletion whereas a key coordinator has to delete exactly the entry which has been selected with the key.

`void setWriteLock(Transaction writeLock);`
This method can be used to set the write lock of an entry. It should be possible to set the write lock without any checks because the entry is new and no other transaction had the possibility to use the entry before. In order to make sure that a write lock is only set if there are no other locks the lock is only granted if all off the following conditions are fulfilled:

- no delete lock from another transaction is set. If a delete lock from an ancestor is set the lock can be granted. This should not happen unless if the implementation of the coordinator is faulty.
- there is no write lock from another transaction. Again, this should not happen because the entry is new and it can not have been written with another transaction.
- there are no read locks. There should be no read locks if a new entry is written as long as all components behave correctly.

If a write lock can not be granted for a new entry this is an indication for a bug within the coordinator or the core implementation. As described above, a new entry should not have any locks and therefore there shouldn't be any locking conflicts.

`Transaction getWriteLock();`
This method returns the current write lock. If no write lock is set, `null` will be returned.

`boolean hasWriteLock(Transaction tx);`
This method checks if the entry has is locked by the given transaction. If the entry has no write lock or a lock from another transaction `false` is returned. If the lock is from an ancestor the method returns `true`.

`void removeWriteLock();`
This method removes the write lock of the entry. It will not perform any checks, therefore it has to be ensured that the transaction has really finished.

`boolean canBeWritten(Transaction tx);`
This method checks if an entry can be written with the given transaction. It should always return `true` if a new entry is used.

`void addReadLock(Transaction tx);`
This method can be used to add a read lock to an entry. A read lock can be granted if all of the following conditions are true:

- there is no delete lock and
- there is no write lock, or the write lock is from an ancestor of the current transaction.

Read locks do not have to be checked because an entry can be read concurrently by different transactions.

`boolean hasReadLock(Transaction tx);`
This method checks if an entry is currently read by the specified transaction. `True` is returned if the transaction occurs in the list of read locks, otherwise `false`.

`void removeReadLock(Transaction tx);`
This method removes the given transaction from the list of read locks. It does not perform any checks so it has to be ensured that the read lock really exists and shall be removed.

`boolean canBeRead(Transaction tx);`
This method checks if an entry can be read by the specified transaction. An entry can be read if the following conditions are true:

- there is no write lock from another transaction or the write lock is from an ancestor of the passed transaction
- there is no delete lock.

`void checkLocks(Transaction tx);`
Checks if any lock is set. An `EntryLockedException` is thrown if there are locks which are in conflict with tx. The following conditions are checked:

- there is no delete lock or the delete lock is from an ancestor
- there is no write lock or the write lock is from an ancestor
- there aren't any read locks or the read locks are from ancestors

`boolean addOnRollback(Transaction tx);`
This method can be used to check whether an entry shall be available in the coordinator after the rollback of the given transaction. This method checks if:

- the entry has not been written by the transaction and
- it has been deleted by the given transaction

The method returns `true` if an entry has been deleted by tx and therefore it should be available after the rollback of the transaction. However, if the entry has also been written with the transaction, `false` will be returned because the entry shall not be available after the rollback since it has been written and deleted under the same transaction.

`boolean removeOnRollback(Transaction tx);`
This method does exactly the opposite of `addOnRollback`. It checks if an entry shall be removed from the coordinator if the transaction is rolled back. The method checks the following conditions:

- the entry has been written by tx and
- the entry has not been deleted by tx.

If an entry has been written under the transaction is has to be removed on rollback of the transaction. If the entry has been written and deleted by the same transaction no changes are required. Therefore the method returns `false` in this case.

Some examples how these methods can be used by a coordinator developer are given in Section 6. Basically, a coordinator implementer has two possibilities to handle transactions within a coordinator.

- The coordinator executes all operations immediately and does not care for transactions. This means that entries which shall be deleted are deleted from the internal data structure of the coordinator and entries which are written are immediately added to the internal data structures. The coordinator implementer just has to check if the entries can be used by checking the locks. In this case, the commit method can be completely ignored because the internal data structures are already up to date. On rollback, the transaction log (see below), which can be retrieved from the `Transaction` object, has to be analysed and the changes have to be undone. The modification can be undone by analysing the locks of the entries and re-add or remove them from the internal data structure if they have a delete or a write lock (for this, the methods: `addOnRollback` and `removeOnRollback` can be used).
- The coordinator maintains an extra data structure in which the modifications are stored (for each transaction) which have been done during the execution of a transaction. If the transaction is rolled back or committed the changes are undone by analysing the internal data structures.

Which of the two possibilities shall be used depends on the semantics of the coordinator. If a coordinator just realises a special order of the entries the first possibility can be used

because no additional information has to be stored. If the coordinator requires additional meta information the second possibility might be better because the meta information can be stored in the internal data structure and it can be used to rollback or commit the transaction.

Every MozartSpaces transaction has an *transaction log* containing all entries which where modified during the execution of the transaction. The transaction log can be used by coordinators to update their internal data structures when a transaction is committed or rolled back (see Section 6). It is stored within the `Transaction` object and can be retrieved from there. The locks of the entries can be used to retrieve the operations which were executed on the entry:

- read lock is set → entry has been read
- write lock is set → entry has been written
- delete lock is set → entry has been deleted

It is possible that multiple locks are set simultaneously. This is the case when multiple operations have been executed on the entry during execution of the transaction. For example a client could read and afterwards delete an entry. The action which has to be taken on commit or rollback has to be derived from the combination of locks. Table 7 illustrates all possible lock combinations and the actions which have to be taken. In this table `true` means that the lock is set and `false` that it is not set. It is assumed that all locks are set by the same transaction. If the action is `add`, the entry has to be in the container after the commit/rollback and if the action is `delete` the entry must not be available. If the entry only has been read or no operation involving the entry was executed it has to stay unchanged.

entry lock			operation	
read	write	delete	on commit	on rollback
true	true	true	delete	delete
false	true	true	delete	delete
true	false	true	delete	add
false	false	true	delete	add
true	true	false	add	delete
false	true	false	add	delete
true	false	false	–	–
false	false	false	–	–

Table 7: Actions to be taken on commit or rollback

5.4 Transactional Container Operations

Since all container operations (`createContainer`, `lookupContainer` and `destroyContainer`) can be used with a transaction it has to be ensured that this operations fulfils the ACID properties. Additionally, it has to be ensured that a container which has been created by a transaction can be used by the same transaction even if it has not yet been committed, whereas operations of other transactions can not use this container. In MozartSpaces, this is ensured by administrating the containers within a container (as described in Section 2.2.4) called container-container. When a container is created using a transaction, the transaction is used to write the container information into the container which administrates the containers. Therefore, if a container shall be used, the container information can only be read using the same transaction until the transaction is committed. If an operation is executed, the read from the container which stores the containers is performed with the same transaction with which the operation is executed. This simple approach ensures that operations can only be performed on containers whose transaction has already been committed or the operations on these containers use the same transaction.

5.5 Transaction Life-cycle

In this section the methods which are needed to control the life cycle of a transaction are explained. Every transaction has to be created using the `createTransaction` API method. As long as it was not committed or rolled back a transaction can be used to execute operations.

5.5.1 Create Transaction

The creation of a new transaction is handled by the `TransactionManager` component. The only tasks which has to be done is to instantiate a transaction object, create a new unique identifier for the transaction, and to store a reference to the transaction in the `TransactionManager` in order to remember that the transaction has been created on this core instance. The creation of the unique id is done with the code illustrated in Listing 15.

The variables `baseUUID` and `count` are private fields of the `TransactionManager`. They are initialised to `UUID.randomUUID().toString()`[14] and `0` on creation of the Transaction manager. In order to be thread safe the access on the `baseUUID` variable is synchronised. The identifier of a transaction consists of the `baseUUID` object and a long (`count`) value which is increased after the id has been created. This has been implemented because the creation of an `UUID` object takes a very long time. If the `count` variable would

[14]see `http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html`

overflow, a new `UUID` object is created and `count` is set to `0`. This approach ensures that the created identifiers are unique and avoids the overhead of creating a new `UUID` object[15].

```
1  private String baseUUID = UUID.randomUUID().toString();
2  private long count = 0;
3  ...
4  String id;
5  synchronized (baseUUID) {
6    if (this.count == Long.MAX_VALUE) {
7      this.baseUUID = UUID.randomUUID().toString();
8      this.count = 0;
9    }
10   id = this.baseUUID + "--" + this.count++;
11 }
```

Listing 15: Code for creating a new transaction identifier

5.5.2 Commit / Rollback

The sequence diagram in Figure 9 depicts the execution of a successful commit operation within MozartSpaces. This is the execution of the operation with a embedded MozartSpaces instance. If the core would run remotely, the `TransactionTask` would be created by the remote communication layer instead of the `Capi`. The sequence of operation is exactly the same for commit and rollback.

The first thing which happens is the creation of a `TransactionTask` object. As described in Section 2.2 a task extends `Runnable` and can be executed within the thread pool. The adding of the task into the thread pool is not depicted in the diagram in order to keep it simple. When the task is executed, it first executes all global aspects which are registered for `commitTransaction` operations. If all of these aspects answer with OK, a reference to the `TransactionManager` is fetched and the commit operation is executed. The `TransacionManager` first gets a list of containers which have been modified with the transaction which shall be committed. This list is stored within the `Transaction` object. Now, the `TransactionManager` executes the commit operation on all containers which have been modified. The `BlockingLayer` of the container delegates the commit call to the `TransactionLayer`. In this layer, the transaction is checked by calling `initTransaction` and all local aspects are executed using the local aspect manager. If all local aspects answer with OK, the call is forwarded to the `ContainerEngine`. Here, the commit method is called on all coordinators of the container. Since MozartSpaces implements pessimistic transactions the commit operation can not fail. When the commit call returns to the `TransactionLayer` all local POST-aspects are executed. In the `BlockingLayer` all blocking operations are waked up with the method `wakeUpBlocking` because it is possible

[15] The same algorithm is used for creating the entry identifier and the container identifier.

Figure 9: Sequence diagram of a successful commit operation.

that blocking operations can now be fulfilled. Finally, in the `TransactionTask` all global POST-aspects are executed and the result of task is set to `VoidEntry` (see Section 2.2.2). For rollback the message flow is exactly the same, therefore it is not explained here.

6 Custom Coordinators

This section describes how a user can implement custom coordination strategies. In Section 6.1 all necessary classes and methods are described and in Section 6.2 the implementation of all currently implemented coordinators is described.

6.1 API Description

This section describes the interface (see Figure 10) which has to be implemented by a coordinator developer. The class diagram of the most important classes is depicted in Figure 10. All methods are defined in the abstract class `ICoordinator`. The methods which have to be implemented by a coordinator developer are described below. The methods `acquireLocks`, `commitLocks` and `rollbackLocks` which are provided to simplify the transaction handling are described in Section 5.3.1. Since XVSM distinguishes between *implicit* and *explicit* coordinators (as described in Section 1.1.1) two interfaces are provided, `IExplicitCoordinator` and `IImplicitCoordinator`. Depending on whether an implicit or an explicit coordinator shall be implemented the corresponding interface has to be implemented by the coordinator.

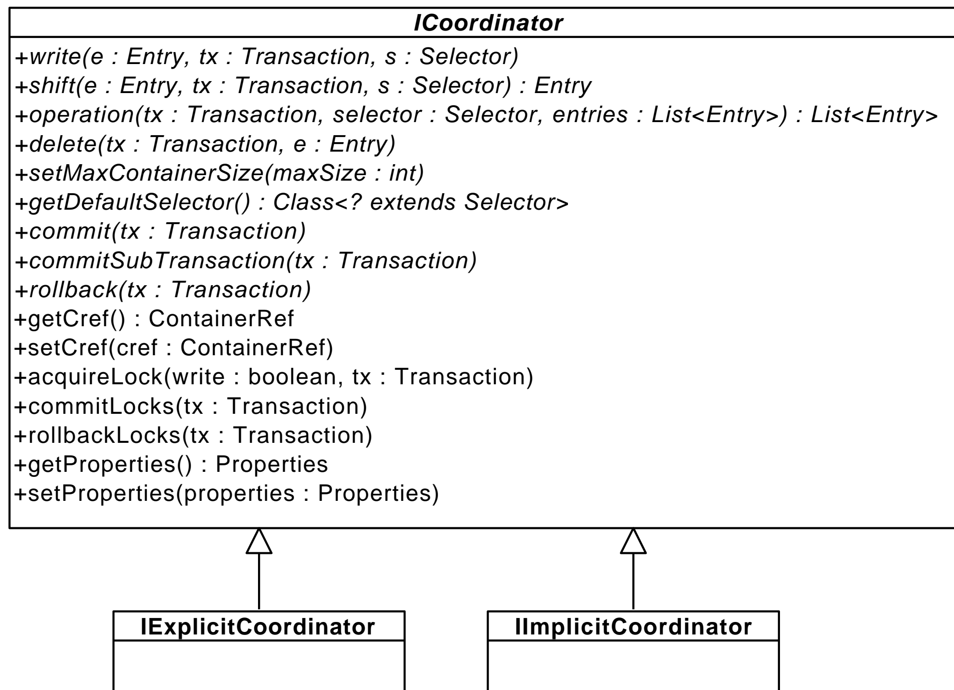

Figure 10: Inheritance hierarchy of the coordinator interfaces

```
public abstract void write(Entry e, Transaction tx, Selector s)
   throws ContainerFullExcpetion, TransactionLockException;
```

This method is called when a new entry (`e`) shall be stored in the coordinator. As described in Section 2.2.3 bulk write operations are split into single write operations in the `TransactionLayer`. Therefore, only one entry is be passed as argument. In addition to the entry, the transaction and the selector which shall be used for writing are passed. If a entry shall be written with multiple selectors for the same coordinator this method is called for each selector separately. The transaction is always set to a valid value (either the transaction passed by the user or an implicit transaction) so it is not necessary to check the status of the transaction. The selector might be set to `null` if the coordinator implements the `IImplicitCoordinator` interface and the user did not provide a corresponding selector. In case of an explicit coordinator this parameter is always an instance of the corresponding selector. Therefore, the object can be cast to an instance of the class which is returned by `getDefaultSelector` (see below). The method declares to throw two exceptions: `ContainerFullException` has to be thrown if the coordinator can not store the entry because there is not enough free space for the new entry left or the position on which the entry shall be stored is already used (for example an entry with the same key already exists, see Section 6.2.6), `TransactionLockException` has to be thrown if another transaction locked the coordinator in a conflicting way (see Section 5.3.1).

```
public abstract Entry shift(Entry e, Transaction tx, Selector s)
   throws TransactionLockException, CannotShiftException;
```

This method is called when an entry shall be shifted into a coordinator. The parameters are equal to those of `write`. When the entry can be written without removing an existing entry, this method does not differ from `write`. When the container is full or one or more entries already exist that disturb the writing, the old entries have to be removed before storing the new one. The entries which have to be removed are determined by the coordinator (for example, a random coordinator can remove a random entry and a FIFO coordinator has to remove the “oldest” entry). The method declares to throw two exceptions: `TransactionLockException` has to be thrown when the coordinator is already locked by another transaction, `CannotShiftException` has to be thrown when the coordinator can not decide which entry has to be removed. For instance, an entry shall be shifted using a key coordinator, the container is full (there are as many entries as the container size allows) but the key on which the new entry shall be written is not yet used. In this case the key coordinator can not decide which entry has to be deleted in order to write the new one. Therefore, the `CannotShiftException` has to be thrown and the shift operation is called again when the shift operation has been called on all implicit coordinators (see Section 2.2.3 for a more detailed description of the execution of the shift operation in the `ContainerEngine`). The entry which has been removed has

to be returned in order to enable the `ContainerEngine` to delete the removed entry from other coordinators. If it was not necessary to remove an entry, `null` has to be returned.

```
public abstract List<Entry> read(Transaction tx, Selector selector,
   List<Entry> entries) throws TransactionLockException,
   CountNotMetException;
```

This method is used to read entries via the coordinator. As with the write operation, the transaction parameter is always set to a valid transaction. In contrast to write and shift, the selector is also always a valid selector because the read operation is only called on a coordinator when the query (specified by the user) contains a selector which corresponds to the coordinator. The amount of entries which has to be returned is stored within the selector (it can be retrieved with `selector.getCount()`). The third parameter is a list of entries which have been selected by preceding coordinators of the same query. Since, multiple selectors in an XVSM query are conjunct using the `AND` operator, the entries which are returned by `read` have to be in the entries which where selected by the previous coordinators. It has to be ensured that the returned entries are not locked by another transaction. This can be realised by using the methods introduced in Section 5.3.1 and Section 5.3.2. When one (or more) entries are locked by another transaction the `TransactionLockException` has to be thrown. The second exception, `CountNotMetException`, has to be thrown when the coordinator does not have enough entries to fulfil the selector.

```
public abstract void delete(Transaction tx, Entry e)
   throws TransactionLockException;
```

This method is used to remove one specific entry from the coordinator's internal data structure (if necessary). The method is called when:

- a `shift` operation has been executed on another coordinator in order to remove the deleted entry from all other coordinators and
- a `delete` or `take` operation is executed. These operations are implemented by first using a read operation to determine the entries which have to be removed, and second use the `delete` operation to remove them afterwards.

The method has to ensure that only entries are removed which are not locked by another transaction. If this is the case, the `TransactionLockException` has to be thrown.

```
public abstract void setMaxContainerSize(int maxSize);
```

This method is called during the initialisation of the coordinator to tell the coordinator the maximum size of the container. This can be used to optimise the internal data

structures. Additionally, the container size is required to determine whether there is enough free space in a coordinator when entries are written.

```
public abstract Class<? extends Selector> getDefaultSelector();
```
This method has to return the class of the selector which corresponds to this coordinator. This is required to determine the correct coordinator for a selector. Please note that it is not possible to have multiple coordinators using the same selector on one container because in this case the `ContainerEngine` could not determine the correct coordinator during the execution of a query.

```
public abstract void commit(Transaction tx)
   throws TransactionLockException;
```
This method is called when the transaction tx is committed. The coordinator implementation has to update its internal data structures.

```
public abstract void commitSubTransaction(Transaction tx)
   throws TransactionLockException;
```
This method is called when a sub-transaction is committed. The coordinator has to delegate the locks on its internal data structure to the parent transaction if necessary. The parent transaction can be accessed using `tx.getFather()`. Please note that it is possible that the parent transaction is rolled back later. Therefore, it has to be ensured that locks are delegated to the parent transaction and not removed.

```
public abstract void rollback(Transaction tx)
   throws TransactionLockException;
```
This method is executed when the transaction tx is rolled back. The coordinator implementation has to update its internal data structures accordingly.

```
public void setProperties(Properties properties);
```
This method is called during initialisation to set the properties of the coordinator. For example, the supported key names in an key coordinator can be initialised that way. A default implementation of this method is provided in `ICoordinator` which returns an empty properties object.

```
public Properties getProperties();
```
This method is used to the get the container properties. It is called when a new container is created to get the properties which are necessary to initialise the coordinator on a remote MozartSpaces instance. A default implementation of this method is provided in `ICoordinator` which returns empty properties.

6.2 Predefined Coordinators

This section describes the currently implemented coordinators. The internal data structures, the implemented coordination strategy and the transaction management are described for each coordinator.

6.2.1 Random

The random coordinator is an implicit coordinator and uses entry locking to process transactions. It plays a special role within MozartSpaces since every XVSM container supports random coordination by default. Therefore, it is not necessary to explicitly tell XVSM to add random coordination during the creation of a container (if an user explicitly specifies a random coordinator during creation the parameter is ignored by the core). Additionally, a random selector can always be used to read (all) entries from a container.

The random coordinator stores all entries in a java hash map, the internal entry[16] identifier is the key and a reference to the entry is the value. The map is used to minimise the access time during deletion of an entry. The *write* operation simply puts the new entry into the hash map and a *delete* operation removes the entry from the map. The locks on the entries are set in the `ContainerEngine`, therefore it is only necessary to check the locks in the coordinator, it is not necessary to update them. The *read* method first checks whether there are entries from preceding coordinators of the query. If this is the case, the read operation only operates on these entries, otherwise all entries are used. Afterwards, the entries are selected randomly until the required amount is selected. During the selection it is checked whether the entries can be read or not using the method `canBeRead` which determines if an entry can be read with a specific transaction (see Section 5.3.2). Only entries are selected which can be read. If there is not a sufficient amount of entries the `CountNotMetException` is thrown, otherwise the selected entries are returned. The *shift* operation checks whether it is necessary to remove an entry before a new one is written. If it is necessary, one entry is selected randomly and removed. The random coordinator ignores the *commit* and *commitSubTransaction* methods because the internal data structure does not have to be updated (entries are inserted into and removed from the hash map immediately). On *rollback*, the transaction log is used to determine which entries have to been removed and which have been written with the transaction. The methods methods `removeOnRollback` and `addOnRollback` (see Section 5.3.2) are used to determine whether a modified entry has to be removed from or put back into the internal data structure.

[16]Internally, every entry has an unique (on the core instance on which it has been created) identifier.

6.2.2 FIFO

The FIFO coordinator is an implicit coordinator and uses container locking to handle transactions. Since this coordinator guarantees the order of the entries it does not make sense to use entry locking. If the oldest entry in the coordinator has an entry lock it has to be waited until the transaction holding the lock is committed or rolled back.

The FIFO coordinator stores all entries in an `ArrayList`. On *write*, the entry is simply added at the end of the list. The *delete* method is ignored, because the read method considers the entry locks during selection. If the oldest entry has a delete lock (which is not compatible with the transaction used for reading) it can not be read and it has to be blocked until the transaction ended. As mentioned before, the read method ignores entries which have a delete lock. If the delete lock is from the same transaction as used for reading the entry is ignored and the next entry is used for reading, otherwise a `TransactionLockException` is thrown. If there are preselected entries from a previous coordinator, the coordinator has to select entries in the correct order from these entries. Therefore, the preselected entries are sorted in FIFO order before the coordinator selects the correct amount of entries. The *shift* operation removes the oldest entry form the coordinator before the new one is written if the container is full. Since the internal data structure is not updated during deletion of entries the coordinator has to handle *commit* and *rollback* of transactions. During a rollback the entries which have been written with the transaction have to be removed from the internal data structure and during a commit the entries which have been deleted with the transaction have to be removed from the internal data structure. This is done by using the transaction log and selecting all entries which have a delete lock. At the end of the commit and rollback method `commitLock` respectively `rollbackLock` in the abstract class `ICoorindator` has to be called to remove the transaction lock from the coordinator.

6.2.3 LIFO

The LIFO coordinator works exactly equal to FIFO. In fact, the coordinator is a subclass of FIFO and only overwrites the write method. Instead of writing entries at the end of the list, new entries are written to the beginning.

6.2.4 Linda

The Linda coordinator is as the random coordinator an implicit coordinator and uses entry locking. The entries are stored in a hash map where the entry identifier is a key and a reference to the entry is the value.

The *write* operation simply adds the entry into the hash map and the *delete* operation removes it. The read operation selects the entries matching the template. A very simple

matching algorithm is implemented currently. All fields of a tuple are compared with the template. Two entries are equal if the following rules apply:

- the types of the entries are equal or the type of the template is set to `null`[17] and
- the value of the entry is equal to the value of the template or the value is set to `null` in the template.

Two tuples are equal if they have the same arity and each entry follows the above rules. A template is represented by a tuple in XVSM. The matching entries are checked if they can be read with the transaction using `canBeRead`. The *shift* operation tries to write the entry; if a `ContainerFullException` is thrown the method tries to determine one entry to delete using the read method, deletes the entry and tries to write again. If no entry can be found for removal, the method throws a `CannotShiftException`. As with the random coordinator nothing has to be done on *commit*. On *rollback*, the entries which have been removed are put back into the hash map and entries which have been written are deleted.

6.2.5 Vector

The vector coordinator is an explicit coordinator and uses container locking. The entries are stored in an ArrayList. The list represents the vector and therefore the entries are stored at the correct position within the list.

On *write* the new entry is added at the correct position in the list. If no position is provided the entry is appended at the end. The *delete* operation is ignored because the entry locks are checked during a read operation. Only, the container locks are acquired. The *read* operation tries to return the entry at the correct position. If the count is greater than one, the entries following the provided position are returned. If preselected entries are passed the read method checks whether the entries are in the list of preselected entries. If this is not the case, `CountNotMetException` is thrown. As mentioned before, the read operation also checks if the entries have a delete lock and can be read with the transaction. The *shift* operation removes the entry which is at the position (at which the new entry shall be written) before it writes the new entry. On *commit*, all entries which have been deleted by the transaction are removed from the list and on *rollback*, all entries which have been written, are removed. Finally, the entry locks are removed using `commitLocks` or respectively `rollbackLocks` (see Section 5.3.1).

[17] `null acts as wildcard`

6.2.6 Key

The key coordinator is an explicit coordinator which uses entry locking. The entries are stored in a hash map containing a hash map. The key of the "outer" hash map is the name of the key and the value is a hash map containing the values of the keys and the entries. Figure 11 illustrates the data structure of the key coordinator. With this

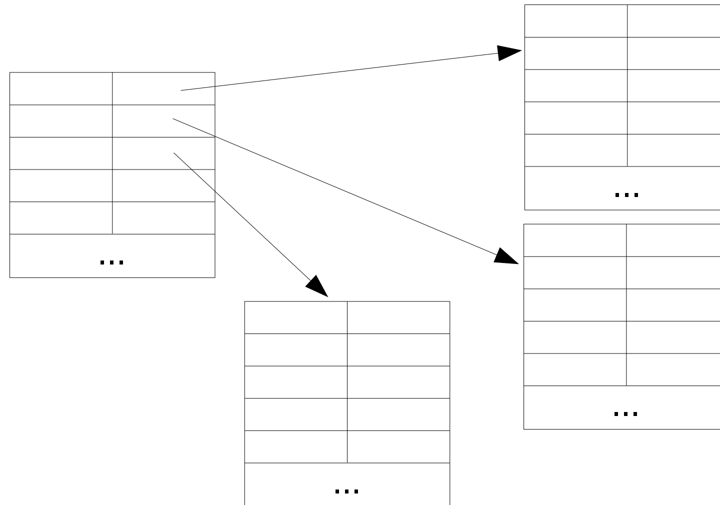

Figure 11: Internal data structure of the key coordinator.

encapsulated hash map it is possible that one key coordinator can handle multiple key names. Additionally, the entries (and the corresponding key selectors) which have been deleted under a transaction are stored in a hash map to make it possible to re-add them if the transaction is rolled back. In order to make the entry deletion faster, a third hash map is used to map the entries to the key selector used for writing. This is necessary because the delete method only has the entry as argument and there is no information on the selector. Without this optimisation it would be necessary to iterate over all entries when one has to be deleted. The supported key names have to be passed during the creation of the coordinator.

The *write* operation puts the new entry into the internal data structures. During *delete*, the entry is removed from the main hash map and stored in the map containing the removed entries. The *read* operation tries to find an entry with the given key in the main hash map. If no such key exists, the `CountNotMetException` is thrown. If preselected entries are provided, the read method has to check whether the entry with the key is in the preselected entries. If this is not the case `CountNotMetException` is thrown. The *shift* operation tries to remove the entry stored with the key on which the new entry shall be stored. If no such entry exists, the `CannotShiftException` is thrown because the key coordinator can not determine the entry which shall be removed. On *commit*, only the map containing the entries deleted with the transaction has to be

removed, the other data structures do not have to be modified since they are up-to-date. If a *sub-transaction is committed*, the entries removed with the sub-transaction are re-added to the map containing removed entries but as if the where deleted by the parent transaction (locks are updated to the father transaction). On *rollback*, entries which have been written under the transaction have to be removed from the internal data structures. This is realised by checking whether the entry log contains entries which are stored in the key coordinator and have to be removed. Additionally, the entries which have been delete have to be added again to the main hash map.

7 XML Protocol

This section describes the XVSM XML protocol. The described protocol is based on initial work by Severin Ecker [24]. The XML protocol enables a platform independent communication in a heterogen network of XVSM nodes. Additionally, it allows firewalls to do a content based filtering which is not possible using a binary serialisation. The parsing of the XML messages is handled by the `XMLMarshaller` class which implements the `IMarshaller` interface (see Section 2.2.8). First, the supported data types and XML constructs are described which are common to all XVSM operations (properties, entries and selectors) and Section 7.6 describes the XML representation of each XVSM operation in detail.

7.1 Supported Data Types

MozartSpaces currently supports the following data types with XML. It is not possible to transport complex data types with the XML protocol because there is currently no default way to serialise objects into XML supported by MozartSpaces. The supported data types reflect the Java primitive types including String and URI:

- Integer
- Boolean
- Character
- Byte
- Short
- Long
- Float
- Double
- String
- URI

If an entry is written which is not an instance of the above classes an exception will be raised by the `XMLMarshaller` implementation.

7.2 XML Serialisation of Properties

Properties are used in every XVSM operation. The aspect context, which can be passed with each operation, is represented as property and selectors, containers and coordinators can have properties as described in the previous sections. XVSM properties always have a key with a unique name and a value. The value of the property is represented by an entry (entry or tuple).

Listing 16 depicts the XML serialisation of an XVSM properties element containing two properties with the names "Vame" and "Value". Properties are enclosed by the `properties` tag. The `property` tag has the mandatory attribute `key`. The value of the

key attribute has to be unique within one properties element. Entries, which are the content of a property, are described in detail in Section 7.4.

```
1 <properties>
2   <property key="Name">
3     <entry type="string">
4       <value>Sepp</value>
5     </entry>
6   </property>
7   <property key="Value">
8     <entry type="string">
9       <value>Meier</value>
10    </entry>
11  </property>
12 </properties>
```

Listing 16: XML representation of properties

7.3 XML Serialisation of Selectors

This section deals with the serialisation of selectors. Since, XVSM allows users to create their own selector (and coordinator), the type of the selector is determined by a class attribute instead of an element. Listing 17 depicts the serialisation of XVSM selectors in XML. The list of selectors is enclosed by the tag `selectors`. Each selector has the mandatory attribute `class` and the optional attribute `count`. The class represents the type of the selector. In the listing, a random, fifo, key and vector selector are shown. A selector can have properties which are used to transport additional information (for example the key of a key selector, or the template of a linda selector). This is shown in the listing with the key and the vector selector. The key selector has three properties, *Name*, *Value* and *Type* and the vector selector has one property which is used to transport the index. The random and the fifo selector do not have additional properties. In MozartSpaces, all selectors have to be located in the java package `org.xvsm.selector`, otherwise the XML marshaller can not instantiate them.

```
1 <selectors>
2   <selector class="RandomSelector" count="1"></selector>
3   <selector class="FifoSelector" count="1"></selector>
4   <selector class="KeySelector" count="1">
5     <properties>
6       <property key="Name">
7         <entry type="string">
8          <value>lalala</value>
9          </entry>
10      </property>
11      <property key="Value">
12        <entry type="string">
13          <value>1232</value>
14        </entry>
15      </property>
16      <property key="Type">
```

```
17        <entry type="string">
18          <value>Integer</value>
19        </entry>
20      </property>
21    </properties>
22  </selector>
23  <selector class="VectorSelector" count="1">
24    <properties>
25      <property key="Index">
26        <entry type="integer">
27          <value>1</value>
28        </entry>
29      </property>
30    </properties>
31  </selector>
32 </selectors>
```

Listing 17: XML representation of selectors

7.4 XML Serialisation of Entries

The serialisation of entries is illustrated in Listing 19. Entries have the mandatory attribute `type` containing the type of the entry. This can be one of the types listed in Section 7.1 or a `tuple`. An entry element can contain a `value` element which stores the value of the entry. In the first entry of the listing the value is “1234”. The value element can be missing (or empty) if the entry represents a template for the linda coordinator. In addition to the value, selectors are supported. This is necessary to make it possible to write entries with different selectors with one bulk write operation. If an entry represents a tuple the attribute `size` has to be provided which can be used to determine the size of a tuple if some fields are null (this minimises the network traffic because empty entries do not have to be part of the XML representation). In this case the entry can have an additional attribute `position` which determines the position of the entry within the tuple (as shown in the listing). Besides “normal” entries and tuples an `exceptionEntry` and a `voidEntry` exist. The `exceptionEntry` represents an exception. It has a mandatory name, in MozartSpaces this is the name of the exception class. The `Description` attributes is optional and can contain a detailed textual description of the exception. `voidEntry` is used as answer for operations returning `void` (see Section 2.2.2). It does not have any attributes.

```
1 <entries>
2   <entry type="string">
3     <value>124</value>
4     <selector class="VectorSelector" count="1">
5       <properties>
6         <property key="Index">
7           <entry type="integer">
8             <value>1</value>
9           </entry>
```

```
10          </property>
11        </properties>
12      </selector>
13    </entry>
14    <entry type="integer">
15      <value>123</value>
16    </entry>
17    <entry type="tuple" size="2">
18      <value>
19        <entry type="string" position="0">
20          <value>eins</value>
21        </entry>
22        <entry type="string" position="1">
23          <value>zwei</value>
24        </entry>
25      </value>
26    </entry>
27    <exceptionEntry name="ContainerFullException"
28      Description="The container is full." />
29    <voidEntry />
30  </entries>
```

Listing 18: XML representation of entries

7.5 XML Serialisation of Coordinators

The serialisation of coordinators is shown in Listing 19. The type of a coordinator is determined by the `class` attribute of the `coordinator` element. In MozartSpaces, coordinators have to be located in the org.xvsm.coordinator package, otherwise the coordinator can not be instantiated. A coordinator can have a list of properties. In the shown example, the key coordinator has a property named “keys” containing a tuple with the name and the type of the key[18].

```
1  <coordinators>
2    <coordinator class="FifoCoordinator"></coordinator>
3    <coordinator class="KeyCoordinator">
4      <properties>
5        <property key="keys">
6          <entry type="tuple" size="1">
7            <value>
8              <entry type="tuple" position="0" size="2">
9                <value>
10                 <entry type="string" position="0">
11                   <value>name</value>
12                 </entry>
13                 <entry type="string" position="1">
14                   <value>string</value>
15                 </entry>
16               </value>
17             </entry>
18           </value>
```

[18]If the key coordinator would support multiple keys, the outer tuple would contain an additional tuple for each supported key name.

```
19          </entry>
20        </property>
21      </properties>
22    </coordinator>
23  </coordinators>
```

Listing 19: XML representation of entries

7.6 Serialisation of Operations in XML

This section describes the XML represention of each XVSM operation. An operation is represented by a `capi` (Core API) XML construct which contains the information of the operation. The `capi` element has one attribute called `answerToContainer` used to transport the URI of the answer container. As described in Section 1.1.2 the answer of an XVSM request is sent to the requester as a write operation into the answer container. If the `answerToContainer` attribute is missing, the result will not be written. Container, aspect and transaction references are always transmitted using their URI representation. All XVSM API operations have corresponding XML representations which are described in the following paragraphs.

Create Container

The `createContainer` element has two optional attributes, the `transaction` contains the URI of the transaction which shall be used for creation and `size` contains the maximum size of the new container. If the transaction is missing an implicit transaction is used for the operation. If the size is missing, -1 is used which represents an infinite size. The `createContainer` element can contain a list of coordinators (see Section 7.5). The XVSM instance on which the container has been created writes an entry containing the URI of the newly created container into the answer container. An example is shown in Listing 20

```
1 <capi xmlns="http://www.xvsm.org"
2   answerToContainer="tcpjava://www.myhost.at/containers/1">
3   <createContainer transaction="tcpxml://www.example.com/transactions/1" size="-1">
4     <coordinators>
5       <coordinator class="FifoCoordinator"></coordinator>
6     </coordinators>
7   </createContainer>
8 </capi>
```

Listing 20: XML representation of a create container operation

Destroy Container

The `deleteContainer` XML element, an example is shown in Listing 21, has the mandatory attribute `containerReference` containing the URI of the container which shall be

removed. Additionally, a transaction can be passed which shall be used to destroy the container. If the transaction is missing an implicit transaction will be used. The answer to a successful deletion of the container is a `VoidEntry` as described in Section 2.2.2.

```
<capi xmlns="http://www.xvsm.org" answerToContainer="tcpjava://localhost:4321/lala">
  <deleteContainer transaction="tcpxml://www.example.com/transactions/1"
    containerReference="tcpxml://www.example.com/containers/1">
  </deleteContainer>
</capi>
```

Listing 21: XML representation of a destroy container operation

Write

The XML representation of a write operation contains the mandatory attribute `containerReference` whic holds the container reference of the target container. Additionally, a `transaction` and a `timeout` attribute can be set containing the transaction and the timeout of the operation. The `write` element contains a list of entries which shall be written into the container. The selectors used for writing have to be stored within the entries. The answer to a successful write operation is a `VoidEntry`. The example, shown in Listing 22, shows a write operation of an ExceptionEntry (see Section 2.2.2). When the write operation is the answer to a request, the answer container is not set in order to avoid that the remote XVSM instance writes an answer to the write operation. Otherwise, an infinite loop would be the result.

```
<?xml version="1.0" ?>
<capi xmlns="http://www.xvsm.org" answerToContainer="tcpjava://localhost:4321/lala">
  <write containerReference="tcpxml://example.com/containers/3334"
    transaction="tcpxml://locahost:9876/transactions/123" timeout="1934">
    <entries>
      <exceptionEntry name="ContainerFullException"
        Description="The container is full." />
    </entries>
  </write>
</capi>
```

Listing 22: XML representation of a write operation

Read

The read operation has the mandatory `containerReference` attribute specifying the container on which the read operation shall be executed and the optional transaction and timeout attributes. The `read` element contains a list of selectors used for reading. The answer to a read operation are the entries fulfilling the selectors. The entries are written solitary into the answer container, i.e., they are not aggregated into a tuple. The example shown in Listing 23 represents a random read operation for ten entries.

```
<capi xmlns="http://www.xvsm.org" answerToContainer="tcpjava://localhost:4321/lala">
```

```
2   <read containerReference="tcpxml://example.com/containers/345"
3    transaction="tcpxml://locahost:9876/transactions/2356" timeout="1934">
4           <selectors>
5             <selector class="RandomSelector" count="10"></selector>
6           </selectors>
7   </read>
8 </capi>
```

Listing 23: XML representation of a read operation

Destroy

The destroy operation, an example is shown in Listing 24, has exactly the same structure as the read operation (described in Section 7.6). The only difference is that the element is named `destroy` instead of `read`.

```
1 <capi xmlns="http://www.xvsm.org" answerToContainer="tcpjava://localhost:4321/lala">
2   <destroy containerReference="tcpxml://example.com/containers/tionContainerID"
3     transaction="tcpxml://locahost:9876/transactions/andfih" timeout="1934">
4           <selectors>
5             <selector class="FifoSelector" count="10"></selector>
6           </selectors>
7   </destroy>
8 </capi>
```

Listing 24: XML representation of a destroy operation

Shift

The XML representation of a shift operation has the same structure as a write operation (described in Section 7.6). The example shown in Listing 25 shifts one entry into a container using the default coordination (random).

```
1 <capi xmlns="http://www.xvsm.org" answerToContainer="tcpjava://localhost:4321/lala">
2   <shift containerReference="tcpxml://example.com/containers/234" timeout="1934">
3     <entries>
4       <entry type="string">
5         <value>eins</value>
6       </entry>
7     </entries>
8   </shift>
9 </capi>
```

Listing 25: XML representation of a shift operation

Create Transaction

The `creatTransaction` element has only one optional attribute `timeout` which represents the timeout of the transaction. If this attribute is missing the transaction will have an infinite timeout. The answer to a create transaction request is an entry which contains the URI of the new transaction. Listing 26 shows an example for the XML of a create transaction creating a new transaction with a timeout of 10000 milliseconds.

```
<?xml version="1.0" ?>
<capi xmlns="http://www.xvsm.org" answerToContainer="tcpjava://localhost:4321/lala">
  <createTransaction timeout="10000" />
</capi>
```

Listing 26: XML representation of a create transaction operation

Commit

The `commitTransaction` element has only one mandatory attribute `transaction` which has to contain the URI of the transaction which shall be committed. The answer to an commit transaction request is a `VoidEntry`. Listing 27 depicts the XML message representing a commit transaction request.

```
<capi xmlns="http://www.xvsm.org" answerToContainer="tcpjava://localhost:4321/lala">
  <commitTransaction transaction="tcpxml://locahost:9876/transactions/1241" />
</capi>
```

Listing 27: XML representation of a commit operation

Rollback

The XML representation of `rollbackTransaction` has the same structure as commit transaction (described above). Listing 28 shows an example of a rollback operation.

```
<capi xmlns="http://www.xvsm.org"
  answerToContainer="tcpjava://localhost:4321/lala">
  <rollbackTransaction transaction="tcpxml://locahost:9876/transactions/andfih" />
</capi>
```

Listing 28: XML representation of a rollback operation

Clear Space

A clear space request only contains the empty element `clearSpace` as shown in Listing 29. The core on which the operation was executed writes a `VoidEntry` into the answer container after successful execution.

```
<?xml version="1.0" ?>
<capi xmlns="http://www.xvsm.org"
  answerToContainer="tcpjava://localhost:4321/lala">
  <clearSpace />
</capi>
```

Listing 29: XML representation of a clear space operation

Shutdown

The `shutdown` element, shown in Listing 30, also does not contain additional attributes.

```
<?xml version="1.0" ?>
<capi xmlns="http://www.xvsm.org"
  answerToContainer="tcpjava://localhost:4321/lala">
  <shutdown />
</capi>
```

Listing 30: XML representation of a shutdown operation

Add Aspect

An `addAspect` element has the `containerReference` attribute containing the URI of the container to which the aspect shall be added and the `type` attribute containing the type of the aspect. If the `containerReference` attribute is missing the aspect will be added globally. The type is mandatory and can be one of the following values:

java The aspect is implemented in java. The `value` element of `addAspect` has to contain the name, including the full class path, of the aspect.

dotNet The aspect is implement in .net.

interop Since the current MozartSpaces version does not yet support aspects implemented in a scripting language, this type has been added for interoperability. If the type is set to "interop" the `value` element has to contain a predefined aspect name which has a clear defined behaviour. Currently, only "NotificationAspect" is supported which is used to realise notifications.

In addition to the `value` element containing the above mentioned values, the `addAspect` element can contain a list of IPoints [41] and the properties of the coordinator. The answer to an add aspect operation is an entry containing the URI of the new aspect. The example shown in Listing 31 depicts the adding of "NotificationAspect" to a container.

```
<capi xmlns="http://www.xvsm.org"
  answerToContainer="tcpjava://localhost:51424/containers/35c875ce-ccd9-44bf-ab44-3
      a4be1e94e31">
  <addAspect type="interop"
    containerReference="TcpXML://localhost:9876/containers/28be947c-4e67-4f11-8820-6
        bea00f8058f">
    <value>NotificationAspect</value>
    <ipoints>
      <ipoint>PostTake</ipoint>
    </ipoints>
    <properties>
      <property key="ncref">
        <entry type="uri">
          <value>
            TcpXML://localhost:9876/containers/b5935de4-8982-45c7-82e1-9c6d1bd3d502
          </value>
        </entry>
      </property>
```

```
17        <property key="cref">
18          <entry type="uri">
19            <value>
20              TcpXML://localhost:9876/containers/28be947c-4e67-4f11-8820-6bea00f8058f
21            </value>
22          </entry>
23        </property>
24      </properties>
25    </addAspect>
26  </capi>
```

Listing 31: XML representation of an add aspect operation

Remove Aspects

The `removeAspect` element, as shown in Listing 32, only has the mandatory attribute `aspectReference` which has to contain the URI of the aspect which shall be removed.

```
1 <?xml version="1.0" ?>
2 <capi xmlns="http://www.xvsm.org"
3 answerToContainer="tcpjava://localhost:51424/containers/35c875ce-ccd9-44bf-ab44-3
      a4be1e94e31">
4   <removeAspect aspectReference="tcpxml://www.example.com/aspects/9" />
5 </capi>
```

Listing 32: XML representation of a remove aspect operation

8 Related Work

There are already a lot of space based computing middle-wares available. This section presents a brief overview over some alternative middle-wares. A detailed comparising of XVSM to other spaced based computing middle-wares is given in [43].

8.1 JavaSpaces

JavaSpaces [26, 36] is a tuple-space implementation developed by Sun Microsystems[19] for the Java programming language. It is part of the Jini [6] framework. Tuples, called entries, are represented by Java object whereas every public field is written to the space (fields with other modifiers are ignored). To identify an object as tuple the interface `Entry` has to be implemented. The standard specifies a simple API for the interaction with the space:

`Lease write(Entry entry, Transaction txn, long lease)` writes a tuple to the space. The transaction under which the operation shall be performed and an lease (specifying the lifetime of the entry) can be specified.

`Entry read(Entry tmpl, Transaction txn, long timeout)` reads one tuple from the space. The template is also representing by a Java object whereas `null` is used as wildcard. Besides the template, a transaction and a timeout can be specified. The timeout declares how long the client is willing to wait for a matching entry.

`Entry take(Entry tmpl, Transaction txn, long timeout)` like read, but deletes the returned entry from the space.

`EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listener, long lease, MarshalledObject handback)` creates a notification on the space. Whenever entries are written matching the template the notification fires (invokes `notify` in the given listener). The lease specifies the lifetime of the notification and the handback is sent to the listener as part of the event notification[20].

In addition, `readIfExists` and `takeIfExists` exists. These methods return `null` if no matching entry exist. The matching and the timeout are handled the same way as for `read` and `take`, except that blocking is only done to wait for a transactional state to settle.

There exist several implementations of the JavaSpaces standard. GigaSpaces [3] is a commercial implementation focused to be an implementation platform for high scalable applications. Outrigger is Sun's open source reference implementation of the JavaSpaces standard. It is shipped with the Jini framework. Another open source implementation is

[19] `http://www.sun.com`

[20] The handback can be used to distinguish which notification fired when the same listener is used for several notifications.

Blitz [2] developed by Dan Creswell.

8.2 TSpaces

TSpaces [35, 14], developed by IBM[21], is a tuple space based middleware with database capabilities. Since it is implemented in Java it enables communication in a network of heterogeneous computers. The API provides similar methods as JavaSpaces, `write` can be used to store tuples, `read` to retrieve and `take` to consuming retrieve tuples from the spaces. The blocking variants of the methods are called `WaitToTake` and `WaitToRead`. Additionally, methods to retrieve all matching tuples are provided (called `Scan` and `ConsumingScan`). In order to expand the set of build-in operators, TSpaces allows an user to dynamically add operators to the spaces. They are called `handlers` in TSpaces terminology. Besides template matching, users can query for tuples based on the value of a given field name, regardless of the structure of the rest of the tuple.

8.3 Corso

Corso [18], developed at the Institute of Computer Languages at the Technical University of Vienna, is based on the idea of virtual shared memories. It allows the sharing of data objects between heterogeneous environments. In contrast to tuple spaces, shared objects in Corso have an unique identifier (called OID) which can be used to create links between objects. This simple mechanism enables a user to build complex and concurrent data structures in the space. Corso has an integrated persistency and transaction mechanism. In addition, it is capable of replicating data object in a network of Corso kernels. The kernel itself is implemented in C, but multiple language bindings are offed (i.e. Java, C++, Ada).

8.4 ActiveSpace

ActiveSpace [1] is a JavaSpace like abstraction on top of a message bus system. It has been designed for implementing applications using a SEDA style architecture. Spaces can be grouped and subspace relations are supported. Similar to JMS systems, the space can operate in two modes, *queue based* where only one consumer receives an object and *publish/subscribe based* where multiple consumers can get the object. Instead of using templates for querying the space, SQL 92 can be used to create a subspace which only contains objects matching the query. ActiveSpace offers a simplified JavaSpace API. Methods for writing (`put`) and consuming objects `take` are provided. `Take` blocks as

[21] `http://www.ibm.com`

long as an object is available or a specified timeout expires. Additionally, ActiveSpace implements a caching mechanism based on JCache [7].

8.5 XMLSpaces

XMLSpaces [48, 47, 16] is an extension to the linda tuple model whereas the fields of a tuple can contain, besides primitive data types, XML documents. Additionally, tuples can contain again tuples which enables the creation of tuple trees. An XML query language can be used to query for tuples. This query language can be XPath, XQuery or any other XML query language. Methods for writing, reading and taking tuples are offered. XMLSpaces supports *FlatTemplates* which only consider the first level of a tuple and *DeepTemplates* which perform a recursive matching of subtuples. XMLSpaces was developed at the TU Berlin and is implemented on top of the .NET framework.

8.6 LIME

LIME (Linda In Mobile Environment) [37, 9] is a tuple space implementation written in Java. The main focus is to support the development of applications in a mobile environment. LIME is capable of distributing the content of a space across multiple mobile components. When mobile components are within range, the content of the space is shared between all components. This distributed space is called *federated tuple space* in LIME terminology. LIME uses so called *tuple locations* to query a specific partition of a federated tuple space. The implementation is independent of the underlying tuple space and mobile agent system. Currently, LigthTS (see Section 8.7) is used as tuple space and the mobile code system μCode[22] is used.

8.7 LighTS

LighTS [8] is a lightweight implementation of a linda tuple space. It focuses only on the main linda features and does not support extensions like, persistency, security, or remote access. These can be implemented around LightTS since it has been designed to be as extensible as possible. Additionally, LightTS provides an abstraction layer, exporting the LightTS API, to enable the transparent replacement of the actual tuple space implementation. Methods for writting, reading and taking single and multiple tuples are provided. Blocking as well as non-blocking variants of the operations are supported.

[22] `http://mucode.sourceforge.net/`

8.8 Mars

Mars (Mobile Agent Reactive Spaces) [20, 19] extends the linda tuple space by programmable reactions. Reactions can perform specific actions based on the accesses made by mobile agents. They can alter the semantic of the agent's access and change the content of the tuple space. In Mars the reactions are stored in a meta-level tuple space which is associated to the tuple space. The Mars API extends the JavaSpaces standard by readAll and takeAll operations which return all matching entries. Reaction can be added to the space which are activated whenever a specific operation is executed. Additionally, a template can be specified which has to match the tuples involved in the operation.

8.9 TuCSoN

TuCSoN (Tuple Centres Spread over Networkds) [40] defines an extension to the linda tuple space. In TuCSoN, each node hosts a local communication space which consists of multiple tuple centres. Tuple centres can be accessed via linda-like operations. The behaviour of each tuple center with respect to the communication event (executed operation) can be modified in TuCSoN. This is realised by so called *Reactions*. A reaction specification language is used to add a reaction to a tuple centre. TuCSoN does not specify which reaction specification language, which tuple type and which tuple matching criterion are used. One possible language for specifying the reactions is *ReSpecT* (Reaction Specification Tuples) [22]. A ReSpecT tuple associates actions to an operation, for example `reaction(Op, Body)` associates the reaction body `Body` to the (space) operation `Op`. Whenever the operation is executed all corresponding reactions are triggered. The body of a reaction can access the information related to the executed operation and access and modify the content of the tuple centre. For the in (consuming read) and rd (read) operations it can also be specified if the body shall be executed before or after the operation in ReSpecT.

9 Implementing a Semantic Space

This Section describes the implementation of a semantic triple space [42] based upon XVSM to demonstrate how the features of XVSM can be used to extend the functionality of MozartSpaces. The specification of the described triple space is similar to the triple space implemented within the TripCom [13] project and can be found in [38].

The space can store semantic information in form of *RDF* [12] triples. Each space can contain an arbitrary number of subspaces. Triples can be written into a specific subspace. If information from a space is queried the information of all subspaces within the queried space are considered, too. Additionally, the space supports a role based security mechanism for every subspace. A detailed description of the security mechanism can be found in [21].

In TripCom, the semantic space consists of several components which communicate with each other using a JavaSpace (see [39]). Table 8 gives a brief description of the components and how they can be mapped to XVSM.

Component	Description	XVSM realisation
Triple Space API	Specification and implementation of the space API	Implementation of the API and mapping to corresponding XVSM API calls.
Triple Store Adapter	Persistent triple storage	Coordinator implementation storing the written triples in an RDF database.
Query Pre-Processor	Optimisation of queries before the execution	Global aspect implementing the optimisation
Distribution Manager	Distribution and finding of triples (and spaces) in a network	Global aspect implementing the distribution of triples.
Metadata Manager	Generating and storing of meta data	Global aspect implementation
Transaction Manager	Support for (pessimistic) transaction management	Local aspect which maps the transaction to the transaction system of the underlying RDF storage.
Security Manager	Implementing the security mechanisms	Local aspects on all containers which check the security attributes.
Mediation Manager	Mediation between different ontologies	Global aspect implementing the mediation.

Table 8: Mapping TripCom components to XVSM

In the XVSM based implementation, every subspace has a corresponding XVSM container having a coordinator which uses an RDF database to store and query triples. In

order to map the relation between the subspaces the XVSM meta containers can be used (see Section 1.1.1). The meta container contains the name of the subspace and the container URI in a key coordinator. Figure 12 depicts the structure of the containers. For simplicity, only the security aspect is shown in the figure. The API implementation can find subspaces, which can be addressed using a URI, by navigating through the meta containers. This approach reacts very flexible to changes in the subspace hierarchy since only the entries from the meta containers have to be modified if a subspace is moved.

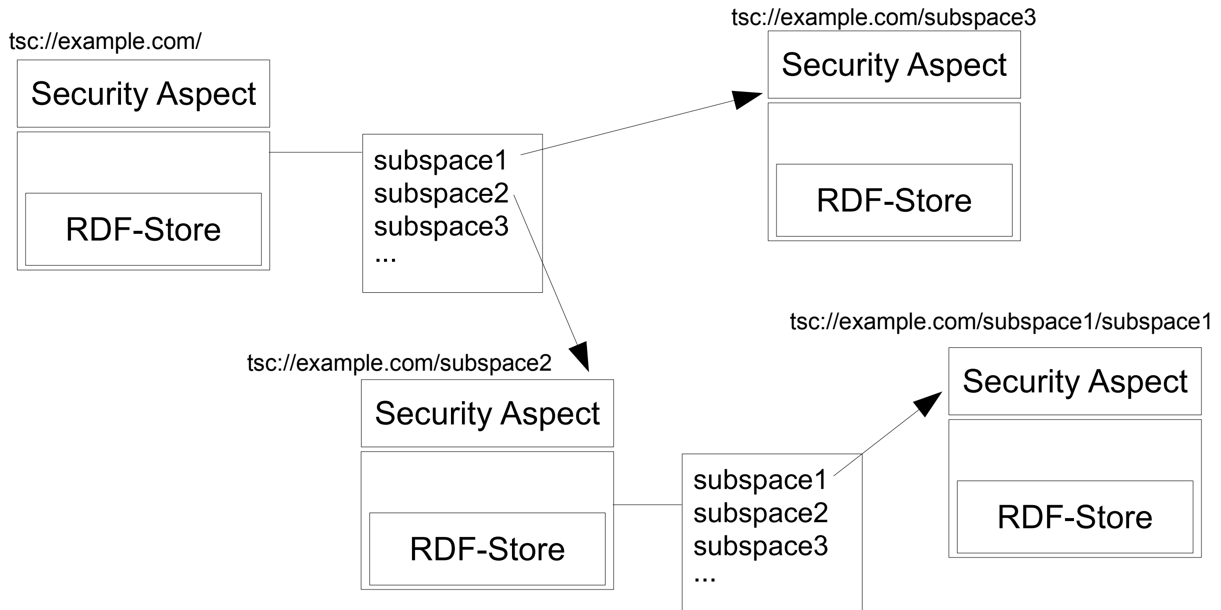

Figure 12: Container structure of the semantic space

Triple Space API maps the triple space API methods to XVSM operations. Additionally, the recursive reading from the subspaces could be implemented in the API but this would require a read operation for each subspace. An alternative approach could be an aspect which intercepts the read operation and handles the reading of the subspaces. This would not require additional read operations and therefore reduce the communication between the API and the XVSM instance.

Triple Store Adapter persistently stores the RDF triples. This component can be implemented as coordinator using an existing RDF database. This approach allows the usage of existing RDF databases instead of implementing a new storage mechanism. In addition, the query engine of the database can be used to resolve read operations.

Query Pre-Processor optimises queries before they are executed. An aspect can be used to do this optimisation before a read operation.

Distribution Manager handles the distribution of triples and spaces in a network. In TripCom the distribution manager uses the Distributed Hash Map implementation P-Grid [11] in order to find triples which are located on other spaces. This could also be realised in an XVSM based implementation by using a global aspect which intercepts each read operation and checks whether the triple can be found in the DHT or not. If the reading from the local space is not necessary (because the triple was found on a remote space), the aspect can use the aspect answer *SKIP* to not execute the operation on the container.

Metadata Manager handles the generation and storage of meta information. Examples for this meta data are: which triple has been written by which user, who read which triples, etc. This behaviour can also be implemented using an aspect. The meta data aspect could write all the meta data of a space into a designated container or into separate containers for each subspace.

Transaction Manager takes care of the transaction management within a space. This could be realised on top of XVSM with an aspect which maps the transaction on transactions of the underlying storage mechanism.

Security Manager handles the access to data. The component has to check whether a user has appropriate access rights for the space and the subspaces. If this is not the case, the aspect can stop the execution of an operation using the `NOT_OK` aspect answer. The security access rules can be stored in one container which is shared among all aspect instances to enable a central management of this information.

Mediation Manager takes care of mediation between different ontologies. This can also be realised using an aspect doing the necessary mediation on every operation.

Implementing most components in form of an XVSM aspect has the advantage that these components can be added and removed to the semantic space as needed. For example, the security aspect can be removed if security is not needed, which would improve the performance of the system. The only mandatory components are the semantic coordinator and the implementation of the triple space API.

10 Future Work

Processing of tasks. The SEDA based run-time environment described in Section 2.1 currently stores tasks in normal Java queues. Instead of these queues XVSM containers could be used. This would enable the prioritisation of tasks by using a specialised coordinator. Additionally, the distribution of a single MozartSpaces instance on multiple machines could be supported by sharing the containers containing the tasks.

The handling of blocking operations could be improved. As described, all blocking operations are waked up when the state of the space/container changes (on write of new tuples and commit/rollback of a transaction). A method could be implemented which selectively wakes up those operations which have the best change to be fulfilled after the change. For example, the selectors or the required locks could be used to select the operations.

Aspects. Currently, aspects can only be executed sequentially in the same order as they have been added and the operation which triggered the aspect is delayed until all aspects have been finished. This could be improved by providing a mechanism to execute aspects asynchronously and allowing to specify a priority which is used to define the order of execution.

Transaction management. The transaction mechanism which is described in Section 5 stores the transaction log in Java lists. This could be improved by storing the transaction log in a MozartSpaces container. This would allow a user to use the transaction log the same way as every other container. In addition, the transaction log could be persisted without the need of an additional persistence mechanism.

The current implementation of MozartSpaces does not provide any deadlock detection mechanisms. The only way to avoid deadlocks is to use transactions with a finite timeout. A deadlock prevention/detection mechanism could be implemented by analysing the active transactions, the currently emitted locks and the locks which have to be acquired next.

XML protocol. A drawback of the current implementation of the XML protocol is that it is not possible to directly serialise complex data types into XML. A serialisation mechanism for Java object could be implemented which can create XML code from every Java object (and vice versa).

Extensible tuple model Research results, such as the "extensible tuple model" (XTM) [25], which defines a formal basis of XVSM, should be analysed with respect to their benefits for MozartSpaces and implemented if appropriate.

11 Conclusion

This thesis describes the implementation of MozartSpaces, the java reference implementation of XVSM. It is a space based computing middle-ware which besides a distributed shared data storage offers plug-able aspects and an extensible coordination mechanism.

The SEDA based MozartSpaces core environment has been described which consists of multiple loosely coupled components implementing the MozartSpaces features. The implementation of the configuration mechanism and the meaning of the configuration parameters has been described. A description of all exceptions and their distribution through the software has been given. The implementation of the fine grained transaction mechanism, which can be used with entry and container locking, has been detailed. Also the possibility to extend MozartSpaces with custom coordination mechanism by implementing a coordinator has been explained and the implementation and semantics of all currently implemented coordinators have been shown. The XML based XVSM protocol has been described which enables a platform independent communication between XVSM instances. Finally, the implementation of a semantic space on top of XVSM has been described to illustrate the possibilities of MozartSpaces.

XVSM and MozartSpaces are still under development. Information on ongoing work can be found on `http://www.xvsm.org` and `http://www.mozartspaces.org`. From the latter, MozartSpaces, which is distributed under an open source license, can be downloaded. Additional information on space based computing can be found on `http://www.spacebasedcomputing.org`.

References

[1] Activespace project web pate, 2008. http://activespace.codehaus.org/Home.

[2] Blitz project web page, 2008. http://www.dancres.org/blitz.

[3] Gigaspaces web page, 2008. http://www.gigaspaces.com.

[4] Jabber web page, 2008. http://www.jabber.org.

[5] Java weg page, 2008. http:www.java.com.

[6] Jini project web page, 2008. http://www.jini.org.

[7] Jsw 107: Jcache - java temporary caching api note = http://jcp.org/en/jsr/detail?id=107, 2008.

[8] Lights web page, 2008. http://lights.sourceforge.net/.

[9] Lime project web page, 2008. http://lime.sourceforge.net/.

[10] Mozartspaces project web page, 2008. http://www.mozartspaces.org.

[11] P-grid web page, 2008. http://www.p-grid.org.

[12] Resource description framework (rdf), 2008. http://www.w3.org/RDF/.

[13] Tripcom project web page, 2008. http://www.tripcom.org.

[14] Tspaces project web page, 2008. http://www.almaden.ibm.com/cs/TSpaces/.

[15] Xcospaces weg page, 2008. http://www.xcoordination.com.

[16] Xmlspaces.net web page, 2008. http://www.ag-nbi.de/research/xmlspaces.net/.

[17] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[18] Johann Blieberger, Johann Klasek, and Eva Kiihn. Ada binding to a shared object layer, 1999.

[19] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. 1 how to coordinate internet applications based on mobile agents, 1998.

[20] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Reactive tuple spaces for mobile agent coordination, 1998.

[21] Davide Cerri, Francesco Corcoglioniti, Hans Moritsch, Jacek Kopecky, and Christian Schreiber. Early prototype of the security and trust infrastructure. Technical report, TripCom, FP6 - 027324, 2008.

[22] Enrico Denti, Antonio Natali, and Andrea Omicini. On the expressive power of a language for programming coordination media. In *In Proceedings of the 1998 ACM Symposium on Applied Computing (SAC'98*, pages 169–177, 1998.

[23] Schahram Dustdar, Harald C. Gall, and Manfred Hauswirth. *Software-Architekturen für Verteilte Systeme*. Springer Verlag GmbH, 2003.

[24] Severin Ecker. Communication protocols in xvsm-design and implementation; master thesis; technical university vienna, insitute of computer languages, 2007.

[25] eva Kühn, Richard Mordinyi, and Christian Schreiber. An extensible space-based coordination approach for modeling complex patterns in large systems, 2008.

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

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

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

[29] David Gelernter. Multiple tuple spaces in linda. In *PARLE '89: Proceedings of the Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, pages 20–27, London, UK, 1989. Springer-Verlag.

[30] Alfons Kemper and André Eickler. *Datenbanksysteme - Eine Einführung, 5. Auflage*. Oldenbourg, 2004.

[31] E. Kühn. *Virtual Shared Memory for Distributed Architecture*. Nova Science Publishers, 2001.

[32] E. Kühn, M. Beinhart, and M. Murth. Improving data quality of mobile internet applications with an extensible virtual shared memory approach. In *IADIS WWW/Internet 2005 Conference, Lisbon, Portugal*, October 19-22, 2005.

[33] E. Kühn, J. Riemer, and L. Lechner. Xvsmp/bayeux: A protocol for scalable space based computing in the web. *Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2007. 16th IEEE International Workshops on*, 0:68–73, June 2007.

[34] E. Kühn, J. Riemer, R. Mordinyi, and L. Lechner. Integration of xvsm spaces with the web to meet the challenging interaction demands in pervasive scenarios. *Ubiquitous Computing And Communication Journal (UbiCC), special issue on "Coordination in Pervasive Environments"*, 3, 2008.

[35] Tobin J. Lehman, Stephen W. Mclaughry, and Peter Wyckoff. Tspaces: The next wave. In *Hawaii Intl. Conf. on System Sciences (HICSS-32*, 1999.

[36] Sun Microsystems. Javaspaces service specification, v1.2.1, 2002. `www.sun.com/software/jini/specs/js2_0.pdf`.

[37] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A middleware for physical and logical mobility. *icdcs*, 00:0524, 2001.

[38] Martin Murth, Gerson Joskowicz, eva Kühn, Dario Cerizza, Davide Cerri, David de Francisco, Alessandro Ghioni, Reto Krummenacher, Daniel Martin, Lyndon Nixon, Nuria Sanchez, Brahmananda Sapkota, Omair Shafiq, and Daniel Wutke. Triple space reference architecture. Technical report, TripCom, FP6 - 027324, 2007.

[39] Lyndon JB Nixon, Daniel Martin, Daniel Wutke, Martin Murth, Elena Simperl, Reto Krummenacher, Brahmananda Sapkota, Zhangbing Zhou, Hans Moritsch, Christian Schreiber, Omair Shafiq, German Toro del Valle, Davide Cerri, and Vassil Momtchev. Platform api specification for interaction between all components. Technical report, TripCom, FP6 - 027324, 2008.

[40] Andrea Omicini. Coordination of mobile agents for information systems: the tucson model. 6th ai*ia convention, 1998.

[41] Michael Pröstler. Design and implementation of mozartspaces, the java reference implemention of xvsm - timeout handling, notitcations and aspects; master thesis; technical university vienna, insitute of computer languages, 2008.

[42] J. Riemer, F. Martin-Recuerda, Y. Ding, M. Murth, B. Sapkota, R. Krummenacher, O. Shafiq, D. Fensel, and E. Kühn. Triple space computing: Adding semantics to space-based computing, 2006.

[43] Thomas Scheller. Design and implementation of xcospaces, the .net reference implemention of xvsm; master thesis; technical university vienna, insitute of computer languages, 2008.

[44] Gernot Starke and Stefan Tilkov. *SOA - Expertenwissen; Methoden, Konzepte und Praxis serviceorientierter Architekturen.* 2007.

[45] Andrew Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd ed.)*. Prentice Hall, 2007.

[46] P. Th, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe.

[47] Robert Tolksdorf and Dirk Glaubitz. Coordinating web-based systems with documents in xmlspaces. In *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*, pages 356–370, London, UK, 2001. Springer-Verlag.

[48] Robert Tolksdorf and Dirk Glaubitz. Xmlspaces for coordination in web-based systems. In *WETICE '01: Proceedings of the 10th IEEE International Workshops on Enabling Technologies*, pages 322–327, Washington, DC, USA, 2001. IEEE Computer Society.

[49] M. Welsh and D. Culler. Overload management as a fundamental service design primitive. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 63–69, New York, NY, USA, 2002. ACM.

[50] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.

Appendices

A Interface Descriptions

commit	Commit a transaction.
Transaction tx	the transaction which shall be committed.
getSize	Get the maximum size of the container.
currentSize	Get the current size of a container. I.e., the current available number of entries.
read	Read entries from the container.
Transaction tx	The transaction which shall be used to read the entries. `null` if an implicit transaction shall be used.
List<Selector> selectors	The selectors which shall be used to read the entries.
int retrycount	The numbers of times the operation has been tried to be executed (0 if it is the first one).
Properties aspectContext	The aspect context passed by the client.
rollback	Rollback a transaction.
Transaction tx	The transaction which shall be rolled back.
shift	Shift entries into the container
List<Entry> entries	The entries which shall be shifted into the container.
Transaction tx	The transaction which shall be used to shift the entries. `null` if an implicit transaction shall be used
Properties aspectContext	The aspect context passed by the client.
take	Take entries from the container.
boolean isDelete	Since delete operations are mapped to take operations internally this parameter indicates if a delete or a take operation is executed in order to execute the correct aspects.
Transaction tx	The transaction which shall be used to take/delete the entries. `null` if an implicit transaction shall be used
List<Selector> selectors	The selectors which shall be used to take/delete the entries.

int retrycount	The numbers of times the operation has been tried to be executed (0 if it is the first one).
Properties aspectContext	The aspect context passed by the client.
write	Write entries into the container.
List<Entry> entries	The entries which shall be written.
Transaction tx	The transaction which shall be used to write the entries.
int retrycount	The numbers of times the operation has been tried to be executed (0 if it is the first one).
Properties aspectContext	The aspect context passed by the client.
getCref	Get the container reference of this container.
setCref	Set the container reference of this container.
ContainerRef cref	The new container reference. This method is only used during the initialisation of the container. The container reference does not change during the life-cycle of a container.
addAscpects	Add an aspect to the container.
List<IPoint> p	The list of IPoints on which the aspact shall be added.
IAspect aspect	The aspect implementation which shall be added.
Properties aspectContext	The aspect context passed by the client.
removeAspect	Remove an aspect from the cotainer
IPoint p	The IPoint from which the aspect shall be removed.
IAspect aspect	The aspect which shall be removed. Note that the equals method has to be accordingly in order to identify the equality of two aspects even when the object reference changed (because of serialisation).
Properties aspectContext	The aspect context passed by the client.
addCoordinator	Add a coordinator to the container.
Class<? extends Selector> s	The selector class to which the new coordinator belongs. Note that `c.getDefaultSelector` has to return the same class.
ICoordinator c	The coordinator implementation which shall be used.

Table 9: ITransactionLayer interface

commit	Commit a transaction.
Transaction txn	The transaction which shall be commited. Here, the transaction is never `null`.
commitSubTransaction	Commit a sub-transaction (sub-transaction are explained in section 5).
Transaction txn	The sub-transaction which shall be commited.
rollback	Rollback a transaction.
Transaction txn	The transaction which shall be rolled back, Here, the transaction is never `null`.
getSize	Get the current size of the container.
read	Read entries from the container.
Transaction tx List<Selector> selectors	The transaction which shall be used for reading. The selectors which shall be used for reading.
write	Write an entry into the container.
Entry entry Transaction tx	The entry which shall be written. The transaction which shall be used for writing.
getCoordTypefromSelector	Get the coordinator implementations for a selector class.
Class<? extends Selector> s	The selector class for which the coordinator shall be returned.
shift	Shift an entry into the container.
Entry entry Transaction tx	The entry which shall be inserted into the container. The transaction which shall be used for shifting the entry.
take	Take entries from the container.
Transaction tx List<Selector> selectors	The transaction which shall be used for taking the entries. The selectors which shall be used for taking the entries.
getCref	Get the container reference for this container.

setCref	Set the container reference for this conatiner.
ContainerRef cref	The new container reference.
addCoordinator	Add a coordinator to the container.
Class<? extends Selector> s	The selector class to which the new coordinator belongs.
ICoordinator c	The implementation of the new coordinator.
getCoodinators	Get all coordinators which are managed by this container engine.
currentSize	Get the current size of this container.

Table 10: ContainerEngine interface

createContainer	Creates a new container.
Transaction tx	The transaction which shall be used to create the container. If an implicit transaction shall be used this is `null`.
String name	The name of the new container. If the container shall not have a name this parameter is `null`.
int size	The size of the new container. If the container shall have an infinite size the constant `IContainer.INFINITE_SIZE` (which is set to -1) is passed.
createContainer	Creates a new container.
Transaction tx	The transaction which shall be used to create the container. If an implicit transaction shall be used this is `null`.
String name	The name of the new container. If the container shall not have a name this parameter is `null`.
int size	The size of the new container. If the container shall have an infinite size the constant `IContainer.INFINITE_SIZE` (which is set to -1) is passed.
ICoordinator... coordinators	An array with the coordinators which shall be supported by the new container.
deleteContainer	Deletes a container.

Transaction tx	The transaction which shall be used to delete the container. If an implicit transaction shall be used this parameter is `null`.
ContainerRef cref	The container reference of the container which shall be destroyed.
getContainer	Get the IContainer implementation of a container.
Transaction tx	The transaction which shall be used. If an implicit transaction shall be used this parameter is `null`.
ContainerRef cref	The container reference for which the container implementation shall be returned.
getContainer	Get the container reference of a named container.
Transaction tx	The transaction which shall be used. If an implicit transaction shall be used this parameter is `null`.
String name	The name of the container whose container reference shall be returned.
getContainer	Get the IContainer implementation of a meta container.
ContainerRef metaCref	The container reference of the meta container which shall be returned.
getMetaContainer	Get the meta container which belongs to a container.
Transaction tx	The transaction which shall be used. If an implicit transaction shall be used this parameter is `null`.
ContainerRef cref	The container reference of the container whose meta container shall be returned.
getAllContainer	Get all containers which exist on the MozartSpaces instance.
Transaction tx	The transaction which shall be used. If an implicit transaction shall be used this parameter is `null`.
getContainerContainer	Get the container which is used to administrate the containers.

Table 11: ContainerManager interface

getInstance	Get the singleton instance of configuration manager.
removeInstance	Remove the singleton instance of the configuration manager.
init	Initialise the configuration manager.
String configurationFile	The path to the configuration file which shall be used for initialisation. The path can be absolute or relative to the working directory.
setStringSetting	Set the value of a configuration parameter.
String key String value	The name of the configuration parameter which shall be set. The value to which the configuration parameter shall be set.
getStringSetting	Get a setting which is a String.
String key	The name of the configuration parameter which shall be returned.
getIntegerSetting	Get a setting which is a Integer. If the value can not be parsed with `Interger.parseInt` an exception might be thrown.
String key	The name of the configuration parameter which shall be returned.
getLongSetting	Get a setting which is a Long. If the value can not be parsed with `Long.parseLong` an exception might be thrown.
String key	The name of the configuration parameter which shall be returned.
getBooleanSetting	Get a setting which is a Boolean. If the value can not be parsed with `Boolean.valueOf` an exception might be thrown.
String key	The name of the configuration parameter which shall be returned.

Table 12: ConfigurationManager interface