



FAKULTÄT FÜR **INFORMATIK**

Design and Implementation of MozartSpaces, the Java Reference Implementation of XVSM Timeout Handling, Notifications and Aspects

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering / Internet Computing

eingereicht von

Michael Pröstler

Matrikelnummer 0304384

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

Developing distributed applications is a difficult process. Several issues like synchronization, coordination or scalability have to be considered. Middleware systems are able to deal with several of those issues and enable faster and easier development. The space-based computing paradigm describes an approach to realize next generation middleware systems.

The goal of this work is to describe mechanisms for next generation middlewares by means of the eXtensible Virtual Shared Memory (XVSM) protocol. Therefore it describes the semantics and the benefits of these mechanisms, especially concentrating on the issues of timeouts, aspects and notifications. This work also describes the implementation of these features within MozartSpaces, the Java reference implementation of XVSM, in detail.

Zusammenfassung

Die Entwicklung von verteilten Applikations ist ein nicht trivialer Prozess. Verschiedenste Faktoren, wie Synchronisation, Koordination oder Skalierbarkeit müssen dabei beachtet werden. Middleware Systeme lösen einige dieser Probleme was zu einer schnelleren und einfacheren Entwicklung führt. Das Space-based Computing Paradigma beschreibt Ansätze für solche Middleware Systeme.

Das Ziel dieser Arbeit ist es, diese Ansätze für eine "next generation Middleware" an Hand von dem eXtensible Virtual Shared Memory (XVSM) Protokoll zu beschreiben. Dabei wird auf die Semantik und Vorteile der jeweiligen Mechanismen eingegangen. Diese Arbeit konzentriert sich vorallem auf Timeouts, Aspekte und Notifikationen. Darüber hinaus wird die Implementierung dieser Mechanismen innerhalb der MozartSpaces, der Java Referenzimplementierung von XVSM, im Detail erläutert.

Acknowledgements

First of all, I want to thank my supervisor Dr. eva Kühn, who introduced me in the fundamentals of the space-based computing paradigm. Without her ideas and her technical knowledge the realization of this master thesis would not have happened.

I also want to thank my parents, who supported me in every situation before, during and still after my studies. Without their support and the chances they offered me to go my way, I would not have become the person I am today.

Additionally my thanks go to all my colleges, who shared a great time with me during studying and afterwards. Especially I want to mention here Richard Mordinyi, who always supported us during the implementation and with whom we had a great time during the last semesters. The second person I want to thank here is Christian Schreiber, who went with me through the whole studies.

At last I want to thank all my friends by speaking in the words of Tennessee Williams:

"Life is partly what we make it,
and partly what it is made by the friends we choose."

Contents

1. Introduction	11
1.1. Motivation	11
1.2. eXtensible Virtual Shared Memory	12
1.2.1. Data Structure	14
1.2.2. Selectors and Coordination Types	16
1.2.3. Operations for Entry Access	17
1.2.4. Aspects	18
2. Semantics and Implementation of Timeout-Handling	20
2.1. Timeouts in General	20
2.2. Operation Timeout	22
2.2.1. Semantics	22
2.2.2. MozartSpace Implementation	23
2.3. Transaction Timeout	29
2.3.1. Semantics	29
2.3.2. MozartSpace Implementation	31
2.4. Transport Timeout	35
2.5. Timeouts in other Space Based Architectures	36
3. Semantics and Implementation of Aspects	39
3.1. Aspects in General	39
3.2. Semantics of Aspects	40
3.3. Aspect Context	43
3.4. Posterior Effects	44
3.5. Significance of Sequence	46
3.6. MozartSpaces Implementation of Aspects	48
3.6.1. Aspect Interfaces and Return Values	48
3.6.2. Aspects General Context	50

3.6.3. User Defined Aspect Context	53
3.6.4. Aspect Management	54
3.6.5. Aspect Execution Process	56
3.7. An Approach for Implementing Access Control by means of Aspects	62
4. Semantics and Implementation of Notifications	65
4.1. Notifications in General	65
4.2. Semantics of Notifications	68
4.3. MozartSpaces Notification Implementation	70
4.3.1. Semantic of the implemented Notification Flavor	70
4.3.2. Notification Implementation Details	72
5. Future Work and Conclusion	76
5.1. Future Work	76
5.2. Conclusion	77
Bibliography	78
A. MozartSpaces API	82
B. Development Process	86
B.1. Version Control	86
B.2. Build System	87

List of Figures

1.1. Space-based Computing Architecture	13
1.2. MozartSpaces Entry Structure	14
1.3. MozartSpace Structure	15
1.4. Container	19
2.1. Timeout	20
2.2. Status Detection	21
2.3. Timeout Detection	26
2.4. Sequence of Timeout Detection	28
2.5. Transaction Network Timeouts	30
2.6. Transaction Timeout Detection	34
3.1. Aspect ReTriggering	44
3.2. Sequential AspectExecution	46
3.3. Bypassing Security	47
3.4. Sequential AspectExecution	47
3.5. Setting a User Defined Context	54
3.6. Aspect Triggering Overview	57
4.1. Polling Mechanism	66
4.2. Notification Mechanism	66
4.3. Number of Requests (Polling & Notification)	67
4.4. Notification Overview	72
B.1. Maven Folder Structure	87

List of Tables

- 2.1. Operation Timeout Support 37
- 3.1. Aspect Triggering 42
- 3.2. Alternative Aspect Triggering 42
- 3.3. Local Aspects Return Values 45
- 3.4. Global Aspects Return Values 46
- 3.5. Aspect Context Available Information 52
- 4.1. Notification Firing 70
- 4.2. Mapping: Notification Target to Join Points 73

Listings

1.1. Generic Atomic Entry	14
2.1. Blocking Decision	24
2.2. IOperationTaskDAO Interface	24
2.3. TimeoutHandler	25
2.4. Update Timeout	27
2.5. Transaction Timeout (Pseudo Code)	30
2.6. Creating Transaction with Timeout	32
2.7. Transaction Timeout Handler	33
2.8. Using Transaction Timeouts	33
2.9. Interface JavaSpace	36
2.10. Interface ITupleSpace	38
3.1. GlobalPoint Class	48
3.2. LocalPoint Class	49
3.3. IAspect Class	49
3.4. AspectContext Class Fields	50
3.5. User Defined Aspect Context Interface	53
3.6. Aspect Manager Execute	54
3.7. Global Aspect Manager Thread Synchronization	55
3.8. Global Pre Aspect Triggering	58
3.9. Global Post Aspect Triggering	59
3.10. ITransactionLayer Interface	59
3.11. ICapi Aspect Methods	60
3.12. Access Control Stub	63
3.13. Access Control Aspect	63
3.14. Adding Access Control	63
4.1. Notification Context	73

4.2. Create Notification Interface	73
4.3. Notification Aspect	74

Acronyms

API	Application Programming Interface
CAPI	Core API
CPU	Central Processing Unit
DAO	Data Access Object
FIFO	First-In First-Out
IDE	Integrated Development Environment
LIFO	Last-In First-Out
LL-API	Low Level-Application Programming Interface
MOM	Message Oriented Middleware
OOP	Object Oriented Programming
REST	Representational State Transfer
SBA	Space Based Architecture
SBC	Space Based Computing
XVSM	eXtensible Virtual Shared Memory

Chapter 1.

Introduction

1.1. Motivation

Creating distributed applications is difficult for most developers. There are many issues, they have to deal with, and there is never a convenient nor a straight-forward solution. Developers have to be aware of many implementation things like synchronization, coordination, scalability as well as technical requirements like firewalls or protocols. Needless to say, that concepts like object oriented programming and component based development also play an important role. The application itself slowly disappears from the developer's sight, while handling those issues. In order to get the developer's focus back on the application, middleware systems were built to ease the development of distributed applications.

First middleware systems were designed to support direct communication between partners. Systems like message passing systems or remote procedure call create a communication channel between exact two applications. Having a number of n partners, they would have to establish $\frac{n*(n-1)}{2}$ connections, when they want to exchange data with all the others. In order to reduce the number of connections, next generation middleware systems used a client/server architecture like a message oriented middleware (MOM), which acts like a communication server. Although this approach reduces the necessary connections to n , it has some serious disadvantages. The entire communication between clients is carried out via the server, which therefore represents a single point of failure. The next generation of middleware systems should provide a hybrid solution, that combines the advantages of both architectures: On the one hand the advantage of direct communication between peers and no single point of failure and on the other hand, a server which handles the communication. The Space-based Computing paradigm[20] fulfills these requirements on a next generation

middleware by running a virtual server, based on a peer-to-peer network. Section 1.2 gives a brief introduction into the mechanisms and the design of XVSM and its Java reference implementation called MozartSpaces in version 1.0.

The remainder of this document is structured as follows: The handling and semantics of different kinds of timeouts are described in chapter 2. In chapter 3 the powerful mechanism of aspects is explained, by providing an overview of aspects in general and concrete implementation details. In chapter 4 notifications and their different meanings and implementation are explained. Conclusion and future work is discussed in the last chapter 5.

1.2. eXtensible Virtual Shared Memory

The “eXtensible Virtual Shared Memory” (XVSM) protocol (see [13] and [12]) realizes the ideas introduced in the previous section of a peer-to-peer based virtual server middleware. For a better understanding of the upcoming chapters, this section will shortly explain the general paradigm of XVSM and the mechanisms of MozartSpaces, the Java reference implementation of XVSM.

XVSM is based on a space based architecture (SBA), which has differences from traditional architectures [20] in the following points (extracted from [20]):

- Everything becomes a service
- The middleware is virtualized
- Clustering is common across all tiers
- Scalability is implicit
- Heavy reliance on memory
- Decoupled communication
- Location transparency
- Heterogeneous by nature

Figure 1.1 shows the result the virtualization, where data and coordination is not handled by a single physical machine, but by all peers. This has the effect that there is no single point of failure and the system can scale by adding additional peers. In addition to these benefits

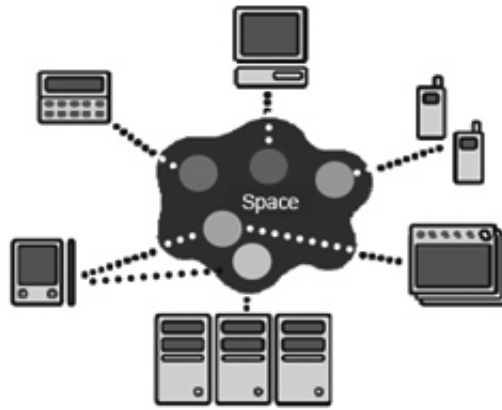


Figure 1.1.: Space-based Computing Architecture

XVSM also provides more flexibility on coordination between applications. Most of existing SBAs like JavaSpaces¹ or XMLSpaces² only support Linda tuple matching as described in [5]. Linda was developed by David Gelernter and is a concept to coordinate distributed applications. A Linda tuple consists of different fields, which can hold every kind of data structure like Linda tuples recursively (see [6] for a detailed description). Searching or reading within a tuple space is done by passing a template tuple with only some values set. Linda tuple matching then is able to return matching tuples.

XVSM extends the coordination possibilities by providing several built-in coordination types, as described in section 1.2.2, and a way to enable developers to create their own coordination mechanisms. These additional coordination features are necessary because Linda tuple matching is often insufficient. Linda tuple matching offers reading by means of a given template, which is a good way for a content based search mechanism. The matching happens because of a passed tuple, which only has some fields set to concrete values. Depending on these set fields, all similar tuples with the same field values will match. Such content based search mechanisms are suited if the necessary additional coordination information can be mapped to the data, e.g.: Reading data structures, which provide a unique identifier. The solution using Linda tuple matching would probably realize this, by adding this identifier to the data structure, and using a template for reading, which only has this identifier set. This approach becomes a problem, when the coordination information is not static, as it is the case when having any kind of order overall data. Simulating a queue or stack is difficult using Linda tuple matching, because the additional coordination information dynamically changes (i.e. the order within the queue or stack) and can therefore not be part of the data structure,

¹<http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/>

²<http://www.ag-nbi.de/research/xmlspaces/>

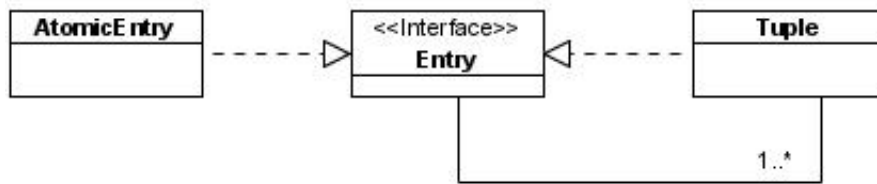


Figure 1.2.: MozartSpaces Entry Structure

when using plain Linda. XVSM offers, beside Linda, several other possibilities to access data to provide flexibility to choose the best fitting way for every use case. In addition to already existing coordination types (random, fifo, lifo, key, vector), XVSM also provides the mechanisms to write higher level coordination types (e.g. poll or auction coordination), in order to enable a tailored and high level communication respectively coordination between applications.

1.2.1. Data Structure

In order to exchange data between peers, an elementary data structure is needed. MozartSpaces therefore uses an entry structure as show in figure 1.2. Whereas an *atomic entry* represents the actual data, a *tuple* is a data object, which consists recursively of other entries, which can either be atomic entries or tuples again.

Atomic entries in MozartSpaces are represented by a generic class, holding data of the generic type to enable to store any type of data. The usage of generic classes in Java allows developers to pass a type in order to grant type safeness and reutilization. Since data in MozartSpaces has to be serialized for the transport in the network the type of the atomic entry has to extend the Java Serializable interface, which is granted by using a Java Wildcards³. The implementation of a very simplified atomic entry class is shown in listing 1.1.

```

1 public class AtomicEntry<T extends Serializable> {
2     private T value;
3
4     public AtomicEntry(T value) {
5         this.value = value;
6     }
7
8     public T getValue() {
  
```

³<http://java.sun.com/docs/books/tutorial/extra/generics/wildcards.html>

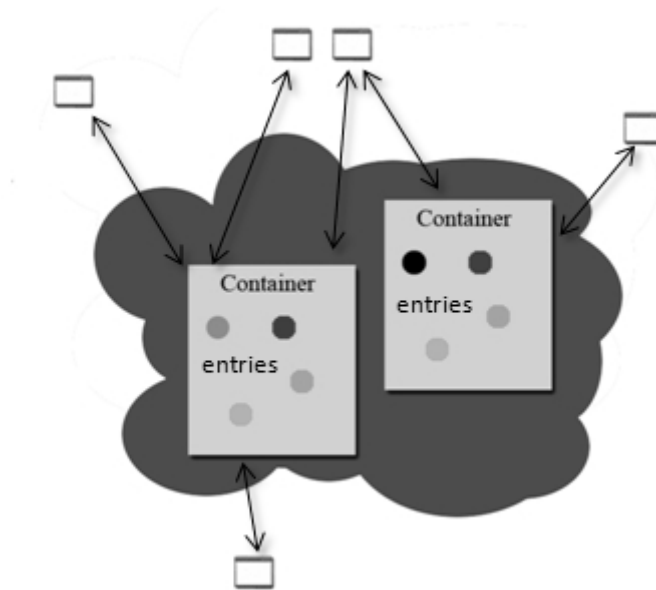


Figure 1.3.: MozartSpace Structure

```

9     return this.value;
10  }
11 }
12
13 AtomicEntry<String> ac = new AtomicEntry<String>("A generic class");

```

Listing 1.1: Generic Atomic Entry

Using this data structure it is possible to write any kind of data into the space. It also supports the writing of binary data into the space. Note that in the current version (MozartSpaces v1.0) data in the space is only held in main memory and no swapping is implemented. This limits the amount of data, which can be held in the space to the heap size size of the Java virtual machine. Entries are stored in a container, which is a data structure within the space. Since entries itself have no identifier, it is not possible to reference a certain entry from the application context, instead the container is responsible for the organization of its contained entries, and the application would ask the container to return an entry. So applications access entries via their containers, as shown in figure 1.3. Applications may delete and create containers and perform entry access or property querying on them, as described in section 1.2.3. Furthermore, containers may have specific properties, such as a limited size, so that they can only store a certain amount of entries.

1.2.2. Selectors and Coordination Types

One of the main advantages of XVSM over existing space-based solutions is the focus on the extensibility of the coordination. Therefore containers, beside storing entries, are able to deal with different coordination types, which can e.g. manage an implicit order, or provide explicit and direct access to entries via keys or labels, or via content matching. Since coordination types can be newly created, there is no limitation of functionality. The following elementary coordination types are supported by MozartSpaces as built-in coordination types:

- random
- fifo
- lifo
- key
- vector
- Linda template matching

Random coordination is the basic coordination that every container is capable of. It does not specify any order but grants random access to the contained entries. The **fifo** coordination type enables the container to act like a queue. It specifies a first-in, first-out behavior.

Using **lifo** coordination, the container represents the behavior of a stack (last-in, first-out) and therefore the inverted behavior of the fifo coordination.

The coordination type **key** enables developers to have a kind of hashmap managed by the container. An entry in the container can be accessed by a specific key. Since entries don't have an identifier the container stores the information of the mapping between the key and the entry. Having **vector** as coordination type enables a container to access entries because of their internal position. When using **Linda** template matching it is possible to read entries out of a container by a given template.

Random, fifo and lifo are referred to as "implicit order" coordination types, because there is no need to specify coordination dependent information when performing operations. The coordination types vector and key are so-called "explicit types". Since the additional information cannot be constructed because of an implicit order, additional information is required to perform actions on these containers. Linda is referred to as a content based coordination type, since the data itself is used for coordination. These coordination types are the built-in ones. In addition developers can implement their own coordination types (e.g. auction or election

coordinators) and use them within their applications. A detailed description of coordination types and their behavior can be found in [11] respectively in [17].

In order to enable coordination types on a container besides the default random type, they have to be explicitly specified when creating the container in the space. A container will always throw an exception if it is not capable of a coordination type, when performing operations on it. In order to access the different coordination types so-called *selectors* exist. Each coordination type has its own selector, which enables the container to decide which coordination shall be used. Selectors can have a Containers always update their implicit coordination types automatically, so that these always hold an up-to-date view of the order of all entries in that container. Explicit types need additional information in order to coordinate them. These informations have to be passed to the container when adding the entries.

1.2.3. Operations for Entry Access

As described in the previous section a container is a structure, which contains entries. The following operations that serve for entry access can be performed on a container:

- read
- take
- destroy
- write
- shift
- notify

In order to get information about the stored entries in a container, the **read** operation is used. When applications want to read from a container, they have to specify the coordination type they want to use, which is done by passing the corresponding *selector* to the container. A selector is additional information for the container, used to coordinate the entries, e.g. a selector could address the first or last entry within a container. Also chains of selectors are supported, which is explained in detail in [18]. Applications can always specify how many entries they want to read by setting the amount within the selector, at which the container will block if it cannot fulfill the request. Depending on the given selector, the container will then perform the operation, according to the coordination type, returning the entries or block until it can fulfill the request.

Taking entries out of a container, extends the read operation by a consuming behavior. According to that, a take operation is a consuming read and can also block, waiting for a certain amount of entries to be taken out of the container.

Destroying entries is like taking them, without being interested in the information about the destroyed entries. Therefore destroy will not return the destroyed entries. It may block when it's not possible to perform the operation.

Adding entries to a container is performed by the **write** operation. Since the container knows how to update an implicit order, it is not necessary to specify further information for that. Otherwise when the application wants to use an explicit coordination type, it has to provide additional information, since the container cannot retrieve this information automatically. An important thing to remember is that the write operation can block. This might be due to several reasons: There might be no space in the container, or because of the semantics of the explicit coordinator (e.g. a given key for an entry is already in the container).

Shift can be seen as a forced write operation. It will first try to perform a normal write operation, but will not block in the case that an entry is already there or the container has already reached its limit. Therefore according to the coordination type of the container, a different entry is deleted to assure that the new one can be written.

The operation **notify** has more an informational character, rather than an access one, but in terms of completeness it is also mentioned here. It informs applications if certain operations are executed on a container. Therefore an application has to specify about which events it wants to be informed. Notifications are near-time, asynchronous and by avoiding polling, an efficient way to track changes of a container. Their detailed semantics and implementation are explained in chapter 4.

For the purpose of understanding the following chapters it is important to know how an operation is executed on a MozartSpace container. A MozartSpace container has different layers, which are responsible for blocking, transaction management and coordination as shown in figure 1.4. Since MozartSpace architecture is based on thread pools according to a staged event driven architecture (SEDA) described in detail in [2], each operation is represented by an according task thread. Tasks are simple Java classes, which extend Runnable, in order to be run by thread pools.

1.2.4. Aspects

Another powerful mechanism for extensibility are *aspects*. Based on the background of aspect oriented programming (described in [15] or [19]), aspects in XVSM can be registered

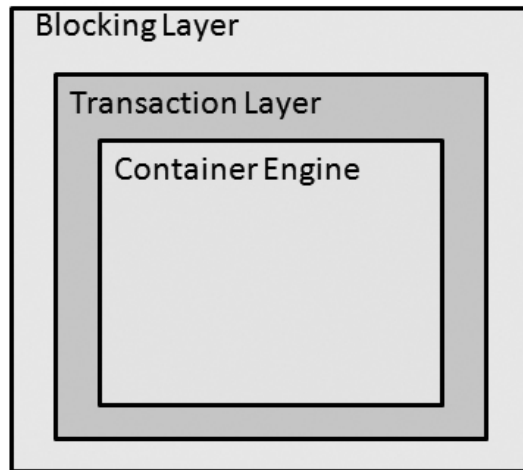


Figure 1.4.: Container

at a container. Aspects are functions, which are triggered by a certain event like an operation on a container etc. In order to specify, when the triggering shall take place, the aspect has to be registered at so-called *interception points*. Possible interception points are *before* and *after* an entry operation. In addition to that, aspects can also be registered on the space, which leads to additional interception points like for creating or deleting a container. Extension points for transactional operations are also available as well as for registering aspects. Whereas aspects on container level are so called local aspects, aspects on the space level are referred to as global aspects. Aspects can also have different kinds of return values, which can affect the whole operation context, where they got triggered as described in section 3.4. There are many application areas for aspects like authorization, encryption and logging. Moreover the MozartSpaces implementation of notifications itself uses aspects as described in section 4. A detailed explanation of the semantics and implementation of aspects is given in chapter 3.

Chapter 2.

Semantics and Implementation of Timeout-Handling

This chapter is dealing with the issue of timeouts and how they are handled. It first explains timeouts and why they are necessary in general. After that, semantics and different kinds of timeouts in XVSM are explained, before heading toward the implementation and its issues in MozartSpaces.

2.1. Timeouts in General

In general a timeout is a specified period of time. As shown in figure 2.1, a timeout Δ is defined between a starting event α and an event β , which will be entered after the timeout Δ .

Timeouts are an important mechanism, when dealing with distributed systems. Starting at the network layer, it might be difficult to determine whether another system is still running or not, due to the existence network failures or networks delays. It might be considered to be

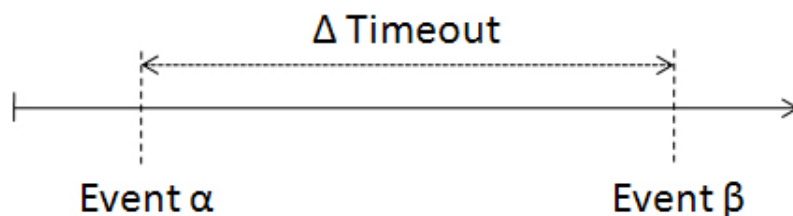


Figure 2.1.: Timeout

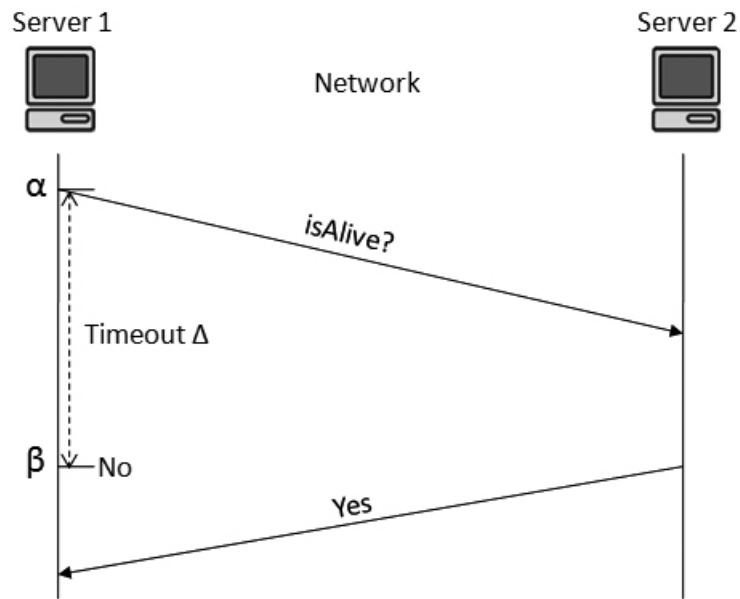


Figure 2.2.: Status Detection

down, only because the network reaction was too slow, which is shown in figure 2.2. Even if the server 2 is still running, the acknowledgment might have been sent too late. Server 1 considers server 2 to be down, since it has not received an acknowledgment within the specified timeout Δ , which is a problem fail-safe systems have to deal with.

When working with network connections, or any type of I/O device, there are two classifications of operations [16]:

Blocking operations: Read or write stalls, operation waits until I/O device is ready.

Nonblocking operations: Read or write attempt is made, operation aborts if I/O device is not ready.

Since XVSM can be seen as a I/O device, operations can block, if the request cannot be fulfilled as shortly described in section 1.2.3. Therefore a timeout is needed to prevent applications waiting infinitely for an operation to be fulfilled and that applications are able to react on these circumstances. These are so called timeout operations and are described in section 2.2. In addition to that, XVSM also introduces timeouts on transactions, which will be explained in section 2.3. The third form of timeouts, which might occur, are network based and therefore not part of the XVSM protocol, but of its Java implementation MozartSpaces instead. This is covered by section 2.4. The following sections explain the purpose and semantics of the timeouts first, before they give a detailed explanation of this implementation in

MozartSpaces.

2.2. Operation Timeout

2.2.1. Semantics

As mentioned above, operations in XVSM can block. Since in many cases it is useful to block only for a period of time, XVSM supports timeouts on the following operations:

read blocks if the read cannot be fulfilled (e.g. no available entries)

take blocks if the take cannot be fulfilled (e.g. no available entries)

write blocks if the container is full or because of semantics of the coordination type (e.g. key already exists)

destroy blocks if the destroy cannot be fulfilled (e.g. no available entries)

Timeouts get specified in milliseconds, whereas -1 indicates an infinite timeout. A timeout of 0 has the semantics of trying once and giving up if the operation cannot be fulfilled. Since the shift operation is not supposed to block, there is no need for a timeout. The timeout of a operation specifies the minimum time, in which XVSM will try to successfully execute the operation, as long as the operation has not yet been successfully executed. The operation will throw an "TimeoutExpiredException" if it cannot be fulfilled within the given timeout. The term "minimum time" might be a little bit confusing, but this is, because XVSM does not assure that the operation will throw the exception exactly after the specified timeout has expired. Section 2.2.2 will explain, why the reference architecture of XVSM tolerates this little impreciseness. Another important thing to know is that the timeout does neither include network time nor time to the first execution of the operation. That means, that the starting event α occurs, when an operation can not be successfully carried out; after that the timeout Δ begins to run. Setting the timeout to zero, will lead in exact by one execution attempt of the operation and will not block.

After the "TimeoutExpiredException" has been thrown, XVSM grants that the according operation has not effected the containers in any way, so that the state of the system remains constant.

2.2.2. MozartSpace Implementation

In this section the implementation of operation timeouts in MozartSpaces is explained in detail. Note that this is the low level API, and therefore there is no method overloading. Since the focus of this section is on timeouts, only timeout relevant parameters are shown and explained. A summary of all parameters and the entire API is given in the appendix A.

The XVSM reference architecture provides for a stage event-driven architecture (SEDA), which is described in detail in [2]. "This architecture allows services to be well-conditioned to load, preventing resources from being overcommitted when demand exceeds service capacity" [2]. As shown in [2] this approach supports massive concurrency and grants a high throughput, especially under load.

MozartSpaces use Java threadpools for the implementation and Java future tasks. A detailed description of the whole architecture is given in [18]. One approach to block operations is of course blocking the according thread. However, this approach would lead to blocking threads within MozartSpaces. Since this is not sufficient and against the concepts of SEDA [2], MozartSpaces does not block operations by blocking threads in the core, but on the client side instead, as described in [18]. As described in section 1.2.3 the implementation of a container consists of three layers:

- Container Engine
- Transaction Layer
- Blocking Layer

In order to block operations, exceptions are raised in the container engine, respectively in the transaction layer. The following two exceptions can be raised:

- ContainerFullException
- CountNotMetException

The first one is thrown if the container is not able to host additional entries. This exception can therefore only be thrown in the case of an unsuccessful write operation. CountNotMetException instead can be thrown if one of the other operations fail. This happens if the specified count of the according selector cannot be fulfilled, as described in [11].

Thus, blocking occurs if a timeout is specified, which is greater than zero or in the case of an infinite timeout. An infinite timeout is represented by -1 respectively a final static variable

INFINITE_TIMEOUT defined in the ICapi interface. The extracted code of the BlockingLayer class 2.1 shows the check on the according conditions, mentioned above.

```
1 private IOperationTaskDAO blockingOperations = new OperationTaskDAO();
2
3 private void blockOperationTask(OperationTask task, Exception e) {
4     if (task.getTimeout() == ICapi.INFINITE_TIMEOUT
5         || task.getTimeout() > 0) {
6         blockingOperations.add(task);
7     } else {
8         task.setResult(e);
9     }
10 }
```

Listing 2.1: Blocking Decision

In order to reschedule blocked operations, they have to be stored. Line 4 in listing 2.1 adds the task to a data access object (DAO) called blockingOperations, an instance of the OperationTaskDAO class. DAO is a software pattern for accessing data source, described in [14]. In the case of a timeout less than zero or equal to zero, the according exception, which was the cause of the failed operation will be returned. This is done, by setting the result in the task object (see line 6). The OperationTaskDAO itself is a simple data structure allowing the operations specified in the interface IOperationTaskDAO shown in listening 2.2.

```
1 public interface IOperationTaskDAO {
2     /**
3      * Add an {@link OperationTask}.
4      *
5      * @param task
6      *         the new task.
7      */
8     void add(OperationTask task);
9
10    /**
11     * Take all blocking operations.
12     *
13     * @return all operations.
14     */
15    List<OperationTask> takeAll();
16
17    /**
18     * Returns all blocking operations.
19     *
```

```

20  * @return all operations.
21  */
22  List<OperationTask> getAll();
23
24  /**
25   * Remove the task.
26   *
27   * @param task
28   *         the OperationTask to remove.
29   */
30  void remove(OperationTask task);
31
32  }

```

Listing 2.2: IOperationTaskDAO Interface

The operations add and remove insert the given instance of OperationTask respectively remove that instance from the DAO. Whereas getAll retrieves all available operation tasks without modifying the DAO, the operation takeAll also retrieves all operation tasks, but removes them from the DAO as well.

After an operation has been successfully executed, all blocked operations on the according container need to be rescheduled. This is because it cannot be decided, which blocked operations were effected by the executed operation, which is a result of the high extensibility of XVSM. As explained in section 1.2.2 users can define own coordination types, which might have slightly different semantics for the different operations. Therefore the method takeAll() of the IOperationTaskDAO has to be called in order to reschedule every task.

In order to detect expired tasks it is necessary to have periodically checks on the timeouts of blocked tasks. In MozartSpaces this is realized by a timeout handler as shown in listening 2.3, which is a simple class within the ContainerManager. The ContainerManager is responsible for the administration of all containers in MozartSpaces as is explained in detail in [18].

```

1 private class TimeoutHandler implements Runnable {
2     /**
3      * The container.
4      */
5     private IContainer c;
6
7     /**
8      * Creates a new TimeoutHandler.
9      */

```

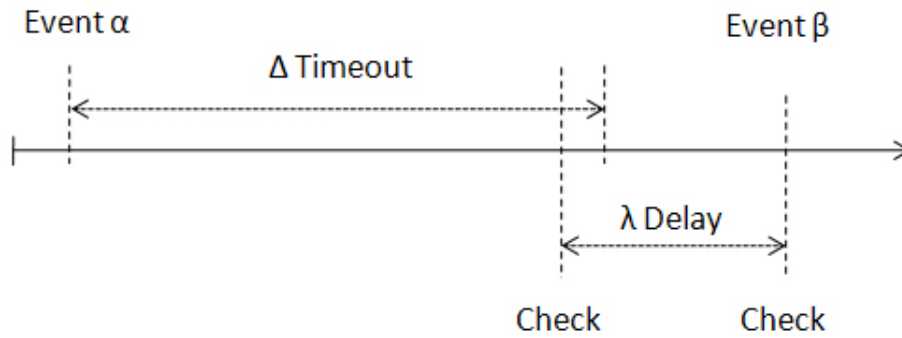


Figure 2.3.: Timeout Detection

```

10     * @param c
11     *         the container to update.
12     */
13     public TimeoutHandler(IContainer c) {
14         this.c = c;
15     }
16
17     public void run() {
18         c.updateTimeouts();
19     }

```

Listing 2.3: TimeoutHandler

In order to call the run method of the timeout handler periodically, it is scheduled by the TimeoutSchedulerPool which internally uses a Java ScheduledThreadPoolExecutor¹. A ScheduledThreadPoolExecutor is a special Java thread pool. It allows to schedule runnables with a fixed delay over and over again. The delay is configurable via a configuration file at deploy time and usually in the range of 50 to 500 milliseconds.

As mentioned above, XVSM does not grant that expired tasks get identified accurate to the millisecond, which is a result of SEDA, respectively of the usage of thread pools. Since timeouts only get checked once within each fixed delay, expired task might get identified shortly behind schedule as shown in figure 2.3. According to the figure 2.3 the maximum time to detect a timed out operation equals the fixed delay of the ScheduledThreadPoolExecutor λ . Therefore MozartSpaces will at least try to successfully execute operations for the set timeout Δ and at most for $\Delta + \lambda - \epsilon, \epsilon > 0$.

As explained, the check of timed out operations is triggered by the periodically scheduled

¹<http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html>

TimeoutHandler and its appropriate run method. The sequence diagram in figure 2.4 gives an overview over the process of the timeout evaluation, which is explained in the following.

The run method, which is called by a thread of the ScheduledThreadPoolExecutor, simply invokes updateTimeouts on the according blocking layer. From there on, all blocked tasks are taken from the BlockingTaskDAO. Since the information about a possible timeout is set in the task, updateTimeout is called on each of the taken tasks. The implementation of updateTimeout method within a task is shown in listing 2.4. It simply calculates and sets the remaining timeout, if it is not infinite. The method has to be aware of the special case that the result of $timeout - timepassed$ in line 5 ends up at -1. Since -1 indicates an infinite timeout the expired task would not be detected. In order to avoid this, the timeout is set to 0, which enables the detection.

```
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 2.4: Update Timeout

After the timeout of the task was updated, the blocking layer checks if the timeout has expired, by calling according method expiredTimeout on the task, which returns a boolean. True is returned if the remaining timeout is less or equal zero, otherwise false is returned. In the case of an infinite timeout the returned value is false. The final step to inform the client about the expired timeout is done by setting a TimeoutExpiredException as a result of the task. This exception will then be raised at the client side, which is described in detail in [18]. Using non-infinite timeouts on operations prevents application from blocking infinitely, whereas the amount of time to wait, should be reasonable. Depending on the semantics of different applications the right timeout has to be well chosen. In general a timeout between 10 and 30 seconds is suitable for most use cases.

XVSM also introduces a further timeout mechanism to avoid infinitely blocking called transaction timeouts; their functionality and semantics are explained in the next section.

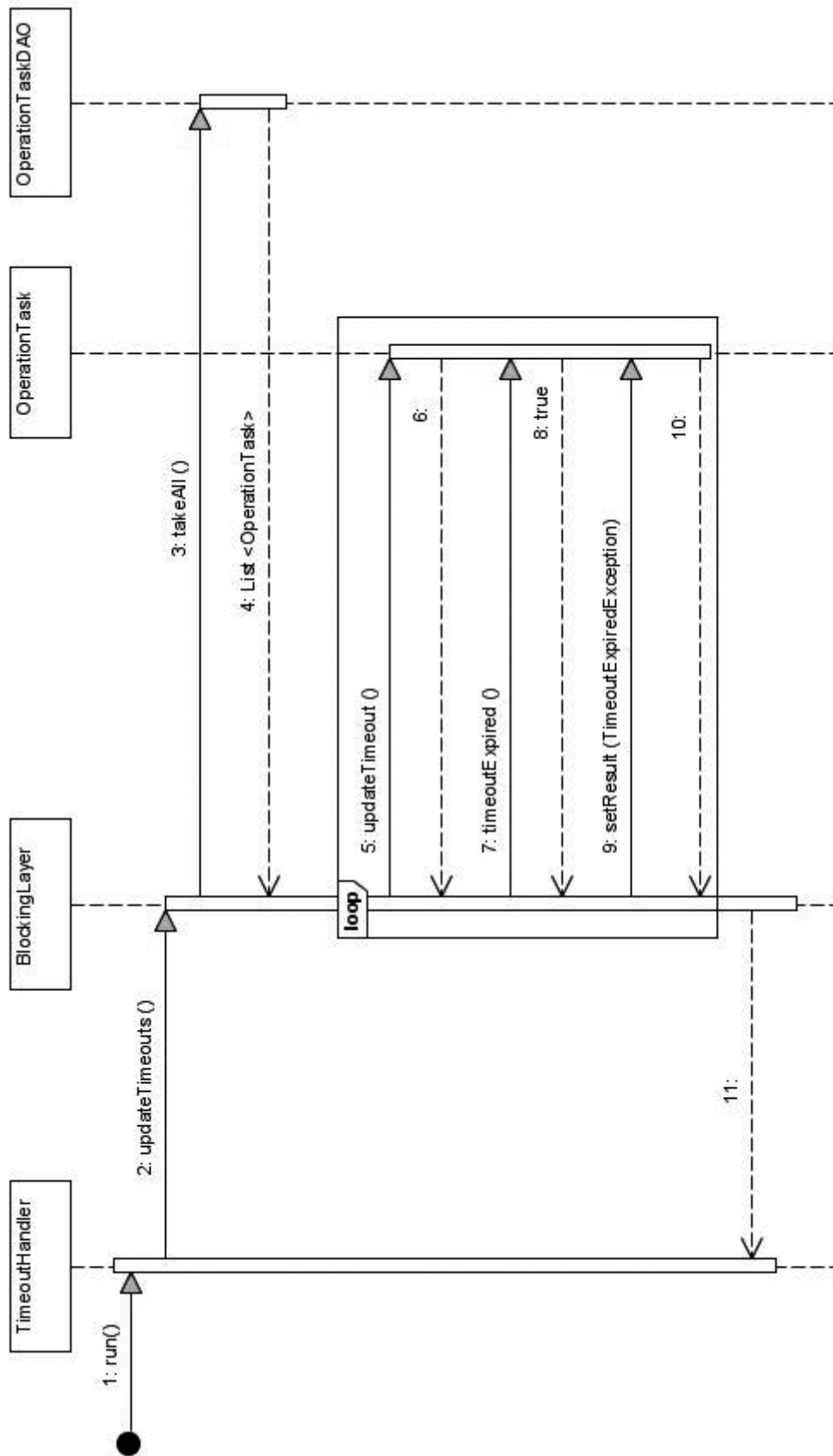


Figure 2.4.: Sequence of Timeout Detection

2.3. Transaction Timeout

2.3.1. Semantics

Besides timeouts on operations, XVSM provides the possibility to set timeouts on transactions as well. This section will shortly explain the term “transaction” in general and introduce the semantics of transaction timeouts. In addition, the necessity and the purpose of these timeouts will be illustrated.

According to Gray, "a transaction is a a collection of operations on the physical and abstract application state" [10]. "The set of operations are based on the Atomicity, Consistency, Isolation and Durability, commonly known as the ACID properties that are to hold true upon establishing a new state" [8]. XVSM supports transactions on all operations mentioned in section 1.2.3. Therefore a created transaction can be passed as a parameter or an implicit transaction will be created instead. An implicit transaction will automatically commit after the execution of its underlying operation. A detailed explanation of transactions and their implementation is given in [18].

Timeouts on transactions specify the time spans of their validity. A timeout of 20 seconds for example implies that the transaction is valid for 20 seconds. Within the 20 seconds the transaction has to be terminated, otherwise it will be rolled back automatically after the timeout has expired. In contrast to operation timeouts, additional operations are involved during the expiration of the timeout. After all operations are terminated the client specifies the end of a transaction by committing it. As a result the commitment has to be recognized by XVSM within the according timeout, otherwise the transaction is automatically rolled back and the commitment will result in an `InvalidTransactionException`, since the transaction is no longer valid. Therefore in contrast to operation timeouts, transaction timeouts include network time and execution time of the clients commit as shown in figure 2.5. Network time refers to the duration of transmitting data between different physical machines over a network, whereas the execution time indicates the span after transmission till execution of the task. This plays a critical role, if applications are distributed over large areas or established on slow networks.

Timeouts on transactions are a powerful way of extending the timeouts on operations. Timeouts on operations are limited to one operation, whereas timeouts on transactions can be seen as an overall timeout. However, operations executed under a certain transaction can still have their own timeout. In order to choose the timeout of a transaction, the timeouts of

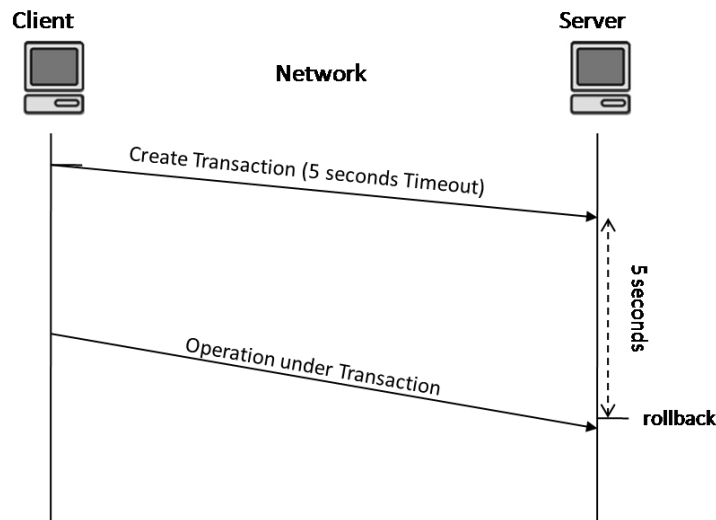


Figure 2.5.: Transaction Network Timeouts

the operations under that transaction have to be considered. Therefore there is not general transaction timeout, that fits for most use cases.

Example: flight and hotel reservation. Requirements are that both reservations are booked or none, under a time constraint of five minutes and that the hotel is only booked if a reservation for the flight exists and vice versa. In XVSM this can be done, as shown in pseudo code in listing 2.5. At first, a transaction with the according timeout is created, followed by writing the reservation of the flight, writing the hotel reservation and finally committing the transaction.

```

1 // 1000*60*5 equates five minutes
2 long timeout = 1000 * 60 * 5;
3 Transaction tx = new Transaction(timeout);
4 write(Airport, FlightReservation, Infinite_Timeout);
5 write(Hotel, HotelReservation, Infinite_Timeout);
6 tx.commit();
  
```

Listing 2.5: Transaction Timeout (Pseudo Code)

Although both write operations (line 4 and 5) might block for an infinite time, the timeout of the transaction will rollback the transaction after five minutes and therefore all non terminated operations under that transaction will end up with an `InvalidTransactionException`. Having only operation timeouts, the time constraint could not be solved optimal. One weak possibility would be to set both operational timeouts to two and a half minute, which would fulfill the time constraint, but would still fail, if the flight reservation blocks for 4 minutes and the hotel reservation does not block for all. As shown by this simple example, timeouts on transactions

are needed in order to fulfill time constraints on more than one operation.

Another purpose of transaction timeouts is the possibility to avoid deadlocks.

"Deadlock is a condition that arises when multiple processes are waiting for resources held by other processes. A process cannot release the resources it holds until it can acquire the resources it is waiting for, but it cannot acquire these resources until another process releases them" [1].

In terms of XVSM, resources can be considered as entries which might be locked by different processes, acquiring entries, locked by other processes and vice versa. XVSM uses pessimistic concurrency control [18] by locking the resources incrementally. [18] describes transactions and locking mechanisms in detail. Introducing timeouts on transactions prevents such deadlocks. Since transactions hold their resources only for the specified timeout, a deadlock will be automatically solved after the timeout of one or more transactions expires. This approach does not need one of the three common strategies [7] for dealing with the deadlock problem: deadlock prevention, deadlock avoidance and deadlock detection and resolution. Nevertheless deadlock detection and resolving will be implemented in the upcoming versions. Transaction timeouts can be seen as a simple automatic deadlock resolution mechanism and since XVSM does not support deadlock mechanisms yet, this is a good way to do that.

2.3.2. MozartSpace Implementation

The implementation of transaction timeouts in MozartSpaces is slightly different from operation timeouts, although some programming patterns (threadpools, tasks) occur in both implementations. One big difference of transaction timeouts is the level of context, where they are defined. In contrast to operation timeouts, timeouts on transaction are not defined on one operation, but involve many different operations. The fact that transactions and as a result their timeouts might be used by different clients, respectively threads is also an important thing to handle as explained in [18].

Using MozartSpaces, developers might define a timeout, when using the API to create a transaction. The according method is specified by the API interface as shown listing 2.6. As mentioned in the javadoc, the parameter "site" addresses the MozartSpace to communicate with. The meaning of aspects is explained in detail in chapter 3.

Since the MozartSpaces implementation is based on thread pools, a runnable class `TransactionTask` is created. This task represents all operations concerning the control of transactions

```

1 /**
2  * Create and start a transaction.
3  *
4  * @param site
5  *       the {@link URI} of the remote core. If null, the
6  *       embedded core is used.
7  * @param timeout
8  *       the time in millicseconds this transaction stays alive.
9  *       The transaction will be rollbacked on expiration.
10 * @throws XCoreException
11 *        if something went wrong during the creation. see
12 *        .getCause() for the reason
13 * @return the created transaction.
14 */
15 Transaction createTransaction(URI site, long timeout) throws
    XCoreException;

```

Listing 2.6: Creating Transaction with Timeout

like create, commit or rollback. As operation tasks, transaction tasks are executed by calling the run method. As far as transaction timeouts are concerned, the only important thing within the run method is the createTransaction method call of the TransactionManager. According to ContainerManager.class, which organizes all containers in the space, TransactionManager.class exists, which is responsible for the whole transaction management. The purpose of the createTransaction method is to create an instance of the Transaction.class and to set the according timeout. In order to reference a transaction, it also stores the transaction in a hash map using the identifier of the transactions as a key. The mechanism how transaction identifiers are created is described in [18]. Another important part is the private class TimeoutHandler, which is part of the transaction manager and shown in listing 2.7. Comparable to the timeout handler of the container manager, a scheduled thread pool is used, to call the run method periodically. The sequence of method calls is shown in figure 2.6. In order to grand exclusiv access to the transaction manager, a read lock is set. After that, updateTimeout is called on every existing transaction. Both, updateTimeout and timeoutExpired methods are implemented in the same way, as the same named methods in the operation task. The first method calculates the remaining time and the second one, returns a boolean, whether the timeout of this transaction has already expired or not. On expiration a new transaction task is created, which rollbacks the according transaction. The implementation of the rollback and the structure of the transaction task are explained in [18]. After the rollback of the transaction, the transaction manager simply deletes the according instance of Transaction.class in its hash map. New operations, which are supposed to be executed under the rollbacked transaction, will end up in an InvalidTransactionException. This approach works fine for new

```

1 private class TimeoutHandler implements Runnable {
2
3     /**
4      * Creates a new TimeoutHandler.
5      *
6      */
7     public TimeoutHandler() {
8     }
9
10    /**
11     *
12     * {@inheritDoc}.
13     */
14    // @Override
15    public void run() {
16        rwLock.readLock().lock();
17        for (Transaction tx : knownTransaction.values()) {
18            tx.updateTimeout();
19            if (tx.timeoutExpired()) {
20                EventProcessingPool.getInstance().execute(
21                    new TransactionTask(TransactionTaskType.ROLLBACK,
22                        tx));
23            }
24        }
25        rwLock.readLock().unlock();
26    }
27 }

```

Listing 2.7: Transaction Timeout Handler

operations, but the handling for blocked operations under a timedout transaction has to be considered. Having a sequence of operations as shown in listing 2.8, it does not suffice to only rollback the transaction.

```

1 Transaction tx = capi.createTransaction(null, 5000);
2 capi.read(cref, ICapi.INFINITE_TIMEOUT, tx);

```

Listing 2.8: Using Transaction Timeouts

Assuming that the read blocks for more than 5 seconds, it would not recognize that the transaction has been rolled back. MozartSpaces solves that issue by simply rescheduling all blocked operations that might be affected by a committed or rolled back transaction. This has to be done anyway, because some blocked operations might now be executed successfully and also a possible way to handle the transaction timeout. The rescheduled operation would then end up in an InvalidTransactionException as well, which is passed to the client. Exception handling is described in [18] in more detail.

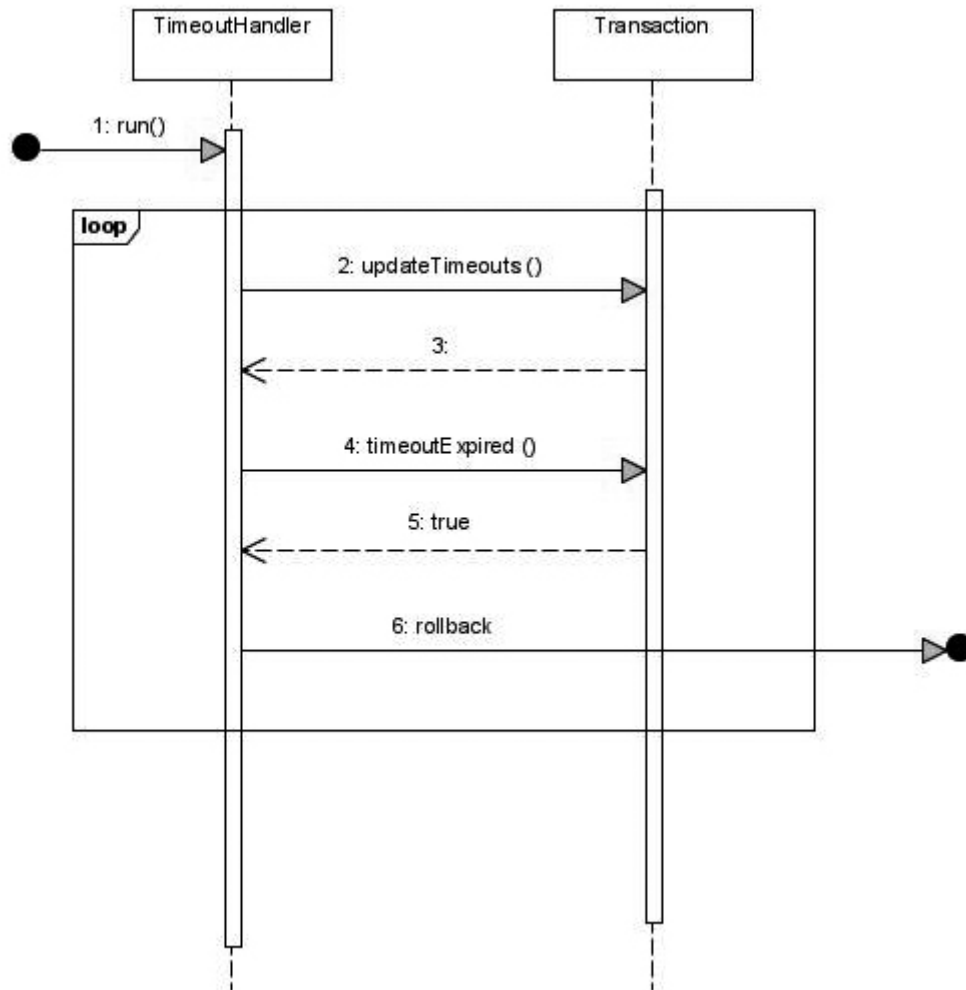


Figure 2.6.: Transaction Timeout Detection

This approach of handling transaction timeouts has the advantage, that it can handle distributed transactions a priori, since transactions are not bound to a session. Distributed transaction can be shared by multiple threads and/or applications. Obviously, if more than one client, respectively threads share the same transaction, they have to coordinate the commitment or rollback of the shared transaction.

The last section of this chapter is dealing with timeouts, which might occur because of the underlying network.

2.4. Transport Timeout

The last sort of timeouts that might occur are network timeouts. Since an embedded running MozartSpace will not connect to any network resource, those timeouts can only occur during a connection to a remote running MozartSpace. Since MozartSpaces uses Java sockets for remote communication, there are only two method call, which might block for a certain period.

- `ServerSocket.accept()`
- `Socket.read()`

Whereas the first one is used for accepting connections, the second one is used for reading from an input stream. Since MozartSpace will always accept connections, the default infinite timeout of the accept method is used. This is the same for the read operation, which will also block for an infinite time, although this should be configurable in the next version.

Since MozartSpaces has different transports and supports different protocols, the different transport mechanism are responsible for their own timeout handling. The protocols implemented in this version support communication over the XVSM XML protocol and by Java serialization. However, both protocols use the default socket timeout settings and support stateless communication.

2.5. Timeouts in other Space Based Architectures

This section deals with the implementation of timeouts in other space-based architectures. Therefor the following two implementations, respectively specifications were studied.

- JavaSpaces Service Specification
- LightTS

The **JavaSpaces Service Specification**² dictates the implementation of timeouts for the four operations: read, readIfExists, take and takeIfExists. Each implementation of this specification has to implement the interface `JavaSpace`. Timeouts only occur in the method headers of the mentioned methods as shown in listing 2.9.

```
1 EventRegistration  notify(Entry tmpl, Transaction txn,
    RemoteEventListener listener, long lease, MarshalledObject handback)
2     When entries are written that match this template notify the given
    listener with a RemoteEvent that includes the handback object.
3
4 Entry  read(Entry tmpl, Transaction txn, long timeout)
5     Read any matching entry from the space, blocking until one exists.
6
7 Entry  readIfExists(Entry tmpl, Transaction txn, long timeout)
8     Read any matching entry from the space, returning null if there is
    currently is none.
9
10 Entry  snapshot(Entry e)
11     The process of serializing an entry for transmission to a JavaSpaces
    service will be identical if the same entry is used twice.
12
13 Entry  take(Entry tmpl, Transaction txn, long timeout)
14     Take a matching entry from the space, waiting until one exists.
15
16 Entry  takeIfExists(Entry tmpl, Transaction txn, long timeout)
17     Take a matching entry from the space, returning null if there is
    currently is none.
18
19 Lease  write(Entry entry, Transaction txn, long lease)
20     Write a new entry into the space.
```

Listing 2.9: Interface `JavaSpace`

In comparison to `XVSM` the `JavaSpaces Service Specification` does not support timeouts in the write method.

²http://www.sun.com/software/jini/specs/js2_0.pdf

	JavaSpaces	LightTS	XVSM
Read Timeout	X		X
Take Timeout	X		X
Destroy Timeout	NA	NA	X
Write Timeout			X

Table 2.1.: Operation Timeout Support

LightTS³ is another space-based implementation but does not implement the JavaSpaces Service Specification. As shown in listing 2.10 the ITupleSpace interface of LightTS does not support timeouts at all.

An overview of the support of timeouts on operations between the implementations of the JavaSpaces Service Specification, LightTS and XVSM is given in table 2.1.

³<http://lights.sourceforge.net/>

```

1 int count(ITuple template)
2     Returns a count of the tuples found in the tuple space that match the
   template.
3
4 String getName()
5     Returns the name of the tuple space.
6
7 ITuple in(ITuple template)
8     Withdraws from the tuple space a tuple matching the template
   specified; if no tuple is found, the caller is suspended until
   such a tuple shows up in the tuple space.
9
10 ITuple[] ing(ITuple template)
11     Withdraws from the tuple space all the tuple matching the template
   specified.
12
13 ITuple inp(ITuple template)
14     Withdraws from the tuple space a tuple matching the template
   specified; if no tuple is found, null is returned.
15
16 void out(ITuple tuple)
17     Inserts a tuple in the tuple space.
18
19 void outg(ITuple[] tuples)
20     Inserts multiple tuples in the tuple space.
21
22 ITuple rd(ITuple template)
23     Reads from the tuple space a copy of a tuple matching the template
   specified.
24
25 ITuple[] rdg(ITuple template)
26     Reads from the tuple space a copy of all the tuples matching the
   template specified.
27
28 ITuple rdp(ITuple template)
29     Reads from the tuple space a copy of a tuple matching the template
   specified.

```

Listing 2.10: Interface ITupleSpace

Chapter 3.

Semantics and Implementation of Aspects

This chapter concentrates on the paradigm of aspects. After an introduction in the evolutionary history, the advantages and concepts of aspect afflicted programming are explained. As already shortly described in chapter 1.2.4, XVSM supports aspects. Therefore the XVSM oriented semantics are explained in detail, followed by a detailed description of the implementation within MozartSpaces. The last section of this chapter shows an approach to add security features to the MozartSpace implementation, by using aspects.

3.1. Aspects in General

Looking back at the history of programming languages new concepts have been developed from generation to generation. After Assembler, which replaced programming in binary code, the concept of procedural, functional and logic oriented programming languages was introduced. In the following, the paradigm of object-oriented programming was introduced, which "is one of the most important contributions to software development in its history" [4]. As described in [9], the benefits of using object-oriented technologies in all phases of the software development process are manifold:

- Reusability of components
- Modularity
- Less complex implementation

- Reduced cost of maintenance

"Each of these benefits will have varied importance to developers. One of them, modularity, is a universal advancement over structured programming that leads to cleaner and better understood software" [9]. [9] also shows the limitation of object oriented programming (OOP) and how aspect oriented programming (AOP) can solve this issues. AOP is mainly based on the homonymous aspects. [4] describes an aspect as follows:

Simply put, an aspect is a particular kind of concern. A concern is any code related to a goal, feature, concept, or "kind" of functionality. An aspect is a concern whose functionality is triggered by other concerns, and in multiple situations. If the concern was not separated into an aspect, its functionality would have to be triggered explicitly within the code related to the other concern and so would tangle the two concerns together. Additionally, because the triggering is in multiple places, the triggers would be scattered throughout the system.

XVSM realizes some parts of AOP. Aspects, concerning XVSM, can be registered at different points as described in section 3.2 and can be triggered by different operations. Aspects in XVSM can be used to implement security as shown in section 3.7 or even more internal functionality like notifications, which is focused in chapter 4. The next section 3.2 gives an overview over aspects in XVSM and concentrates on the semantics of aspects.

3.2. Semantics of Aspects

As described, aspects are triggered by concerns in multiple situations. According to the definition by [4], mentioned in section 3.1, triggering concerns of aspects within XVSM are represented by different operations. They are triggered by operations either on a specific container or on operations related to the whole space. This powerful mechanism enables developers to inject their own code, which is triggered in multiple situations. They can use the aspects "to write code representing crosscutting concerns once and have it appear wherever needed" [15]. In AOP there are so called join points in order to decide, when a certain aspect should be triggered.

A join point is a well-defined location within the primary code where a concern will crosscut the application. Join points can be method calls, constructor invocations, exception handlers, or other points in the execution of a program [9].

In XVSM the join points of AOP are called interception points (IPoints). Interception points, dependent on container operations are referred to as local interception points, whereas interception points on space operations are called global interception points. In addition to that, interception points can be located before or after the execution of an operation. Therefore they are split into pre and post ones.

In XVSM it is also possible to add aspects, based on a local interception point, to all containers. This means, that this originally local aspect is added to every existing container and automatically to newly created ones. Therefore such aspects are also referred to as global aspects, although they still are added at a local interception point. The following local interception points exist in XVSM:

- pre/ post read
- pre/ post take
- pre/ post destroy
- pre/ post write
- pre/ post shift
- pre/ post aspect appending
- pre/ post aspect removing

In addition to the local interception points the following global interception points exist:

- pre/ post transaction creation
- pre/ post transaction commit
- pre/ post transaction rollback
- pre/ post container creation
- pre/ post container destruction
- pre core shutdown

Those interception points can be used to trigger aspects before and/or after the execution of the according operations. The main goal of the interception points for aspect appending and aspect removing is to enable exclusive aspects. This means, that an aspect is able to prohibit the addition of new aspects, respectively prohibit itself to be removed. As it will be described in section 3.7, this is necessary for implementing security via aspects or any other

IPoint/Operation	Read	Take	Destroy	Write	Shift
Pre/Post Read	X				
Pre/Post Take		X			
Pre/Post Destroy			X		
Pre/Post Write				X	
Pre/Post Shift					X

Table 3.1.: Aspect Triggering

Point/Operation	Read	Take	Destroy	Write	Shift
Pre/Post Read	X	X			
Pre/Post Take		X			
Pre/Post Destroy		X	X		X
Pre/Post Write				X	X
Pre/Post Shift					X

Table 3.2.: Alternative Aspect Triggering

mechanism, where an exclusive aspect on a container is needed.

In order to use aspects in XVSM, it is necessary to know, when they are triggered. The triggering of aspects at the different points is shown in table 3.1.

As shown in table 3.1, triggering is really dependent on the operation and not on the according impact. For example a shift operation will only trigger aspects at the shift point, even if the operation itself ended up in a normal write operation. Another approach would be to have aspects triggering on the impact, which would lead to the aspect triggering behavior, shown in table 3.2.

Aspect triggering with the semantics shown in table 3.2 is more complex. Since the focus of this approach is on the impact of operations, an aspect on the destroy point, has to be triggered, whenever something is destroyed. This might be in the case of a shift operation, shifting something out or a take operation. Another problem is, that it is still difficult to say, when an aspect at the shift point should be triggered. Since a shift operation might ends up at a normal write, an aspect at the shift point would not be triggered, although it is registered at the shift point. All in all, it is simply not that straight forward, to decide when an aspect is supposed to be triggered or not.

XVSM semantics of triggering therefore uses the simple and clearly distinguishable approach shown in table 3.1. Anyway, developers can simulate semantics of the second approach by simply writing aspects which are triggered by different interception points.

Another very important thing is, that developers can create stateful and stateless aspects. Stateless aspects will always exhibit the same behavior, regardless of the number of calls etc. In the opposite stateful aspects are able to keep track of previous calls, which might lead to a different behavior, dependent on the actual state of an aspects. The implementation of notifications within MozartSpaces relies on stateful aspects, as it is described in chapter 4. However, the XVSM protocol does not dictate any life cycle behavior, neither for stateless nor for stateful aspects.

The following section 3.3 describes the context of a triggered aspect and the information that is available to that aspect at execution time.

3.3. Aspect Context

In order to make the mechanism of aspects as powerful as possible, XVSM provides all information of the triggering operation to them. All information of the according operation including its transaction are passed to the aspect. An aspect can use this passed context to make logical decisions and to modify this context. A post aspect for example is able to modify the actual result of an operation, whereby the behavior of an operation can be changed completely.

Another important piece of information, which is passed in the context is the number of executions of an operation. This is the result of the possibility that an aspect can be triggered several times by the same operation, which is shown in figure 3.1. Before an operation is really executed, the according pre aspects are triggered. After that, the execution of the operation is performed. As shown in figure 3.1 the operation might be blocked and is rescheduled as described in chapter 2. When the operation gets rescheduled, the aspect is triggered again. Therefore the number of the executions is passed to the aspect, which makes the aspect aware of the fact, that it was already triggered the according number of times.

In addition to that, aspects might need some information exchange with an application. Therefore clients are able to set aspect contexts, which will then be available to the triggered aspect. The transmitted context is represented by a simple key/value pair and is available to every aspect. Section 3.7 provides a use case for a client set aspect context.

In summary an aspect has access to the following information:

- the triggering operation
- all operation dependent information (container reference, etc.)
- number of operation executions
- client set aspect context

The following section 3.4 will explain the semantics of the different return values supported by aspects.

3.4. Posterior Effects

As already mentioned, aspects in XVSM have an additional feature beside the gains of aspect oriented programming described in section 3.1. Instead of just being called, XVSM allows aspects to even affect system behavior after their execution. Therefore aspect are able to return different values, which affect the following course of the operation. In this version of the XVSM the following four values are specified:

- Ok
- Not-Ok
- Reschedule
- Skip

The opposite of the **ok** value, which indicates that everything worked fine during the aspect executing, is the **not-ok** value. This is returned by the aspect, if something went wrong during its execution. If a transaction exists in the context of the aspect, XVSM will rollback the

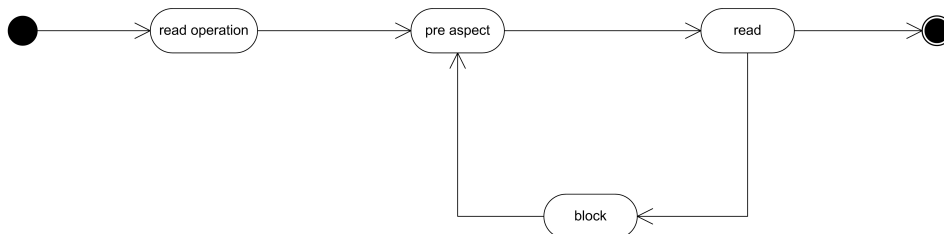


Figure 3.1.: Aspect ReTriggering

Aspect/Return Value	Ok	Not-Ok	Reschedule	Skip
Pre-Read	X	X	X	X
Post-Read	X	X	X	
Pre-Take	X	X	X	X
Post-Take	X	X	X	
Pre-Destroy	X	X	X	X
Post-Destroy	X	X	X	
Pre-Write	X	X	X	X
Post-Write	X	X	X	
Pre-Shift	X	X	X	X
Post-Shift	X	X	X	
Pre-Aspect-Appending	X	X	X	X
Post-Aspect-Appending	X		X	
Pre-Aspect-Removing	X	X	X	X
Post-Aspect-Removing	X		X	

Table 3.3.: Local Aspects Return Values

transaction and an exception is raised at the client.

Reschedule is returned, whenever an aspect wants the operation to be rescheduled, which in terms of XVSM means, that the execution of the operation simply happens an unspecified time later. In order to reschedule the operation within a post aspect, the effects of the operation are rolled back and the whole operation starts from the beginning after some delay.

The last return value is the **skip** value. If a pre aspect returns this value, the following course of the operation is simply skipped including all still not executed pre aspects. Skipping an operation will immediately result in the execution of the post aspects but will not end in an rollback of the transaction as the not-ok value would do. Obviously some values are restricted to pre aspects. Table 3.3 and table 3.4 give an overview over the restrictions of return values when using aspects within XVSM.

As shown in table 3.3, the pre aspects of read, take, destroy, write, shift append and remove aspects are able to return all values. The skip value can only be returned by the pre aspects, since it's not possible to skip an already executed operation. Section 3.5 deals with the issue of aspect sequences and their semantics in XVSM.

Aspect/Return Value	Ok	Not-Ok	Reschedule	Skip
Pre-Transaction-Creation	X	X	X	X
Post-Transaction-Creation	X			
Pre-Transaction-Commit	X	X	X	X
Post-Transaction-Commit	X			
Pre-Transaction-Rollback	X	X	X	X
Post-Transaction-Rollback	X			
Pre-Container-Creation	X	X	X	X
Post-Container-Creation	X			
Pre-Container-Destruction	X	X	X	X
Post-Container-Destruction	X			
Pre-Core-Shutdown	X	X	X	X

Table 3.4.: Global Aspects Return Values

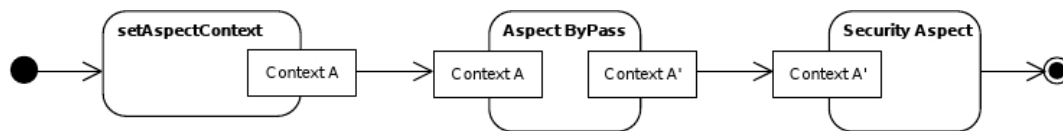


Figure 3.2.: Sequential AspectExecution

3.5. Significance of Sequence

An important thing to know, when dealing with aspects, is that the sequence of the adherence plays a significant role. Although aspects in general are able to work independently, aspect return values in XVSM still provide the ability to affect other aspects (e.g: by returning skip value). Therefore the semantic and the impacts on the sequence of triggering aspects have to be clarified, which is handled in this section. When having more than one aspect appended on a container at the same ipoint, which both can be triggered by the same operation, there are two different approaches:

- Sequential Execution
- Parallel Execution

The **sequential** approach is shown in figure 3.2. This approach will trigger the aspects in the order, in which they were appended. Since the aspect itself is executed within a given context, the context has to be passed from aspect to aspect. As described in section 3.3 aspects are able to modify this context, which might lead to different behavior, if the order of the aspects changes. When implementing security features by using aspects, this is a very

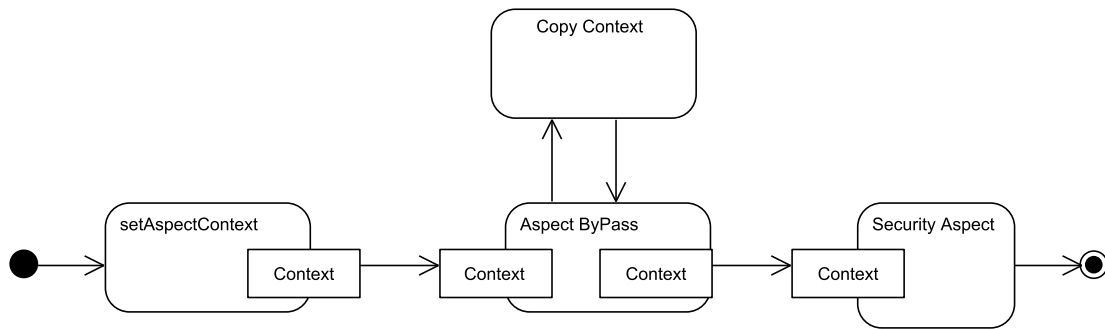


Figure 3.3.: Bypassing Security

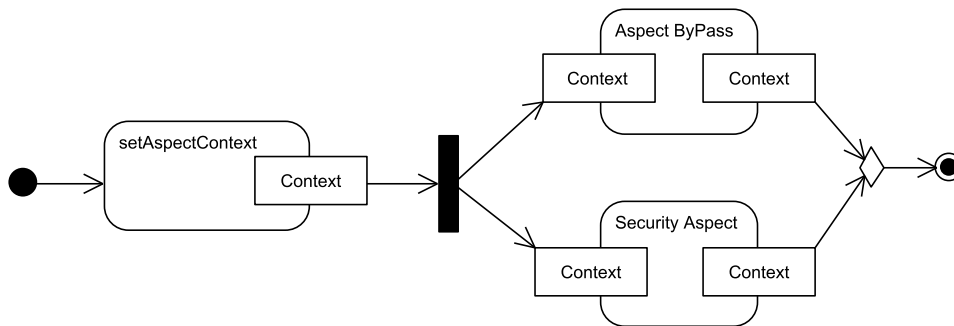


Figure 3.4.: Sequential AspectExecution

important issue. Having a wrong order of triggering would possible cause a bypassing of the security features as shown in figure 3.3.

If the bypass aspect is triggered before the security aspects, it could copy the context with all information to another place or even prevent the execution of the security aspect by returning skip. In order to grant a specific order, additional mechanisms are needed, which is the downside of this approach. A possible solution for this issue is to pass an additional parameter, when adding an aspect. This parameter can represent the position of the aspect. Another approach is the **parallel** execution of aspects, which is shown in figure 3.4. In this approach all aspects are executed in parallel and with the same context. The triggering would the enable the execution of all aspects at the same time. However, this results in a problem of merging the results and the modified context. Since more than one aspects could have changed the results of an operation, it's not possible to decide, which one should be delivered back to the client. Since all aspects at the same time, it's not even possible to prevent an aspects execution by returning skip within another. This is another drawback of this solution, since the return value skip is limited of its power, because it can not prevent other aspects from triggering.

Because of the mentioned disadvantages of the second approach, XVSM chooses the first one.

After the introduction in aspects and their semantics the following section 3.6 describes the implementation within MozartSpaces.

3.6. MozartSpaces Implementation of Aspects

In order to be independent of third parties, MozartSpaces uses the pure functionality of Java 1.5¹ and no specific frameworks like Aspectj².

3.6.1. Aspect Interfaces and Return Values

The package `org.xvsm.core.aspect` contains all relevant classes used for the aspect implementation. Starting at the different interception points, MozartSpaces provides an empty interface `IPoint` which is implemented by `LocalPoint` and `GlobalPoint` in order to differ between local and global aspects. Both implementations of the `IPoint` interface are enumerations to identify the interception points. The classes are shown in listening 3.2 respectively listening 3.1.

```
1 public enum GlobalPoint implements IPoint {
2
3     PreTransactionCreate, PostTransactionCreate,
4
5     PreTransactionCommit, PostTransactionCommit,
6
7     PreContainerCreate, PostContainerCreate,
8
9     PreContainerDestroy, PostContainerDestroy,
10
11     PreTransactionRollback, PostTransactionRollback, PreCoreShutdown
12 }
```

Listing 3.1: GlobalPoint Class

¹<http://java.sun.com/j2se/1.5.0/>

²<http://www.eclipse.org/aspectj/>

```

1 public enum LocalIPoint implements IPoint {
2
3     PreRead, PostRead,
4
5     PreTake, PostTake,
6
7     PreDestroy, PostDestroy,
8
9     PreWrite, PostWrite,
10
11    PreShift, PostShift,
12
13    PreAddAspect, PostAddAspect,
14
15    PreRemoveAspect, PostRemoveAspect
16 }

```

Listing 3.2: LocalIPoint Class

All aspects have to extend the abstract class `IAAspect`. This class specifies the methods shown in listing 3.3 and implements the Java `Serializable` interface. This is because aspects might be transferred through network by Java serialization and therefore they have to be serializable.

The `protected properties` field is only used for remote communication, which is described in detail in [18]. The abstract method `execute`, which has to be written by each aspect, is the most important one. It takes an aspect context and is able to throw one of the following exceptions:

- `AspectNotOkException`
- `AspectRescheduleException`
- `AspectSkipException`

```

1 public abstract class IAAspect implements Serializable {
2
3     protected Properties properties;
4
5     public abstract void execute(AspectContext c) throws
6         AspectNotOkException,
7         AspectRescheduleException, AspectSkipException;

```

```

8  public void setProperties(Properties props) {
9      this.properties = props;
10 }
11
12 public Properties getProperties() {
13     return this.properties;
14 }
15 }

```

Listing 3.3: IAspect Class

These three exceptions are used to implement the XVSM mechanism of aspect return values. Another approach would be to return a structure instead of having a void operation. MozartSpaces chose the exception approach because it fits more easily into its architecture. Since the architecture of MozartSpaces has several layers as shortly described in section 1.2.3, each layer has to process a possible return value and forward the value to the layer above. Implementing this mechanism, by actually returning representative data structures would lead to return values in each method in the progress of an operation. This would also affect operations, which do not want to react on an aspect return value.

As a result of the mentioned drawbacks, MozartSpaces implements the return values by throwing exceptions. Using this approach, methods just have to (re-) throw the exceptions, which then can be caught by the layer above. This also does not prevent methods from return its own data structures as the other approach would do. Each of the exceptions is realized by a simple class, which extends XCoreException, as every XVSM related exception in MozartSpaces does.

3.6.2. Aspects General Context

Beside the possibility to throw one of the specified exceptions, the aspects have unlimited access to the context of their execution. This context is represented by the AspectContext class, which is passed to the aspect as a parameter in the execute method. As explained in section 3.3 the aspect is capable of reading and modifying the given context. In MozartSpaces, an instance of the AspectContext class is passed to the aspect, when it is triggered. The class itself simply contains fields, representing all the information. Every field, shown in listing 3.4 can be accessed by using the according getter and setter methods.

```

1  private LocalIPoint localIPoint;
2

```

```

3  private GlobalIPoint globalIPoint;
4
5  private int retrycount = 0;
6
7  private ContainerRef cref;
8
9  private Transaction tx;
10
11 private List<Selector> selectors;
12
13 private ICoordinator[] coordinators;
14
15 private List<Entry> entries;
16
17 private List<Entry> deleted;
18
19 private List<IPoint> ipoints;
20
21 private IAspect aspect;
22
23 private String containerName;
24
25 private int containerSize;
26
27 private long timeout;
28
29 private Properties aspectContext;

```

Listing 3.4: AspectContext Class Fields

Depending on which operation is triggering an aspect, the fields in the given instance of AspectContext are set to a value or null. This is because some information is only available if a specific operation triggers the aspect. Table 3.5 gives an overview over which attributes are set on which operation.

Every instance of AspectContext has either the localIPoint or the globalIPoint set, which is related to the operation triggering the aspect. Additionally the user defined aspectContext is also added to the instance, which is also explained in this section.

As far as container operations are concerned, the aspects triggered by those operations, have the same information to process. Differences mainly exist between pre and post aspects, since pre aspects can't have the information about the results of an operation. There-

Aspect/Information	localPoint	globalPoint	retryCount	cref	tx	selectors	coordinators	entries	deleted	ipoints	aspect	containerName	containerSize	timeout	aspectContext
Pre Read	X		X	X	X	X								X	X
Post Read	X		X	X	X	X		X						X	X
Pre Take	X		X	X	X	X								X	X
Post Take	X		X	X	X	X		X	X					X	X
Pre Destroy	X		X	X	X	X								X	X
Post Destroy	X		X	X	X	X		X						X	X
Pre Write	X		X	X	X	X		X						X	X
Post Write	X		X	X	X	X		X						X	X
Pre Shift	X		X	X	X	X		X						X	X
Post Shift	X		X	X	X	X		X	X					X	X
Pre Add Aspect	X		X	X	X	X				X	X				X
Post Add Aspect	X		X	X	X	X				X	X				X
Pre Remove Aspect	X		X	X	X	X				X	X				X
Post Remove Aspect	X		X	X	X	X				X	X				X
Pre Container Create	X		X	X	X	X	X			X	X	X	X		X
Post Container Create	X		X	X	X	X	X			X	X	X	X		X
Pre Container Delete	X		X	X	X	X	X			X	X	X	X		X
Post Container Delete	X		X	X	X	X	X			X	X	X	X		X
Pre Transaction Create	X		X	X	X	X									X
Post Transaction Create	X		X	X	X	X									X
Pre Transaction Commit	X		X	X	X	X									X
Post Transaction Commit	X		X	X	X	X									X
Pre Transaction Rollback	X		X	X	X	X									X
Post Transaction Rollback	X		X	X	X	X									X
Pre Core Shutdown															X

Table 3.5.: Aspect Context Available Information

fore the post aspects of read, take, destroy and shift have the additional information about read, written or destroyed entries, which is set in the List<Entry> entries, respectively the List<Entry> deleted attribute.

All aspect related operations such as add or remove aspect, have all the same information. They receive the according aspect itself and information about its triggering points.

The container related operations like create and delete, also receive information about the specific container. Therefore the container name and the container size is also passed within the aspect context in addition to the coordinators of that container.

Aspects triggered before a core shutdown operation, only receive the user set context, since there is no additional information to be passed.

3.6.3. User Defined Aspect Context

Since MozartSpaces support the powerful mechanism of passing attributes to a triggered aspect, the ICapi interface dictates the two methods shown in listing 3.5.

```
1 Properties getAspectContext() throws XCoreException;  
2 void setAspectContext(Properties aspectContext) throws XCoreException;
```

Listing 3.5: User Defined Aspect Context Interface

Those two methods allow users to set and receive Java properties on their interface to MozartSpaces. A Java properties instance is able to store a key value pair, which is used to pass information to a triggered aspect. The Java properties instance is always passed to MozartSpaces, after it was set. Therefore it's the user responsibility to reset properties that should not be passed further. This is done by calling the setAspectContext method, passing null as a parameter. The whole process is shown in figure 3.5. After the user set the aspect context on the ICapi, it is passed to the MozartSpaces core. There, it is enveloped in an according instance of the AspectContext class using the attribute aspectContext as described in section 3.6.2. Since this instance is passed to the triggered aspect, the user defined information can now be processed by the aspect.

The next section will introduce the management of aspects within MozartSpaces, describing the necessary data structures.

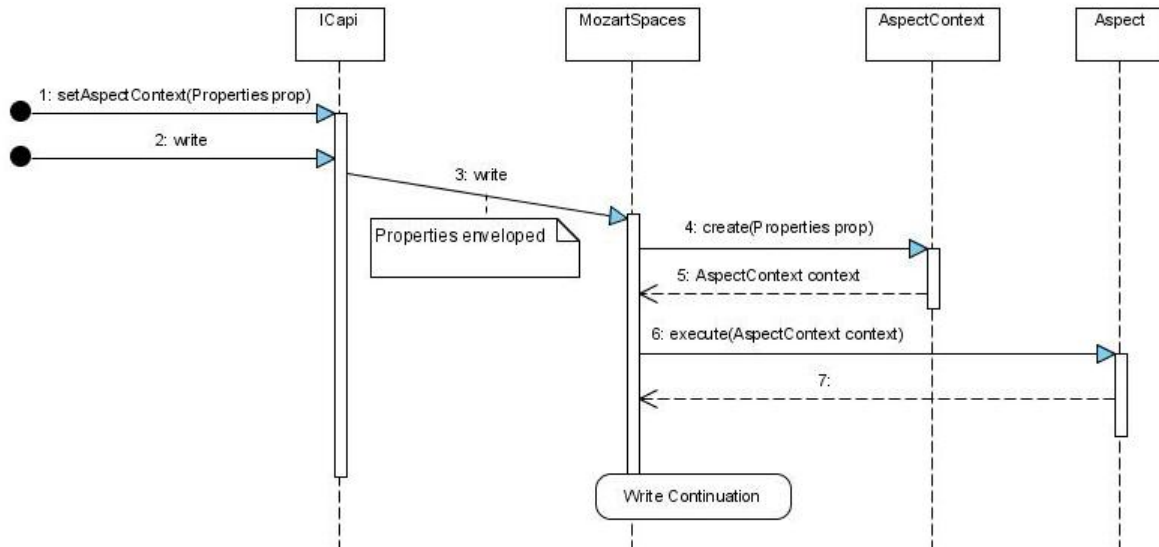


Figure 3.5.: Setting a User Defined Context

3.6.4. Aspect Management

In order to understand the management of aspects in the Java reference implementation, two Java classes need to be considered.

- AspectManager class
- GlobalAspectManager class

The **aspect manager** is part of the transaction layer and is created for each instance of a container. It contains a list of all aspects according to the specific IPoint (`Map<IPoint, List<IAAspect>> aspects`). In order to grant reusability and extensibility MozartSpaces chose to specify the abstract interface for interception points in this class even if global aspects will never be added to a aspect manager at a container. The only method of the aspect manager is the execute method, shown in listing 3.6.

```

1 public void execute(AspectContext c) throws AspectNotOkException,
2     AspectRescheduleException, AspectSkipException {
3
4     List<IAAspect> allAspects = new ArrayList<IAAspect>();
5     IPoint p = c.getLocalIPoint();
6     if (p != null) {
7         allAspects.addAll(this.getAspects(p));
8     }
  
```

```

9     p = c.getGlobalIPoint();
10    if (p != null) {
11        allAspects.addAll(this.getAspects(p));
12    }
13    for (IAspect a : allAspects) {
14        a.execute(c);
15    }
16 }

```

Listing 3.6: Aspect Manager Execute

The method simply takes an instance of the `AspectContext` class and executes added aspects by calling the `execute` method as specified in the `IAspect` interface. As mentioned in section 3.6.1 return values in `MozartSpaces` are handled by the use of exceptions. Therefore the method in the aspect manager just rethrows the according exceptions and will not try to execute outstanding aspects. This is a result of the defined semantics described in section 3.4.

The second class **GlobalAspectManager** is used for every aspect, which is not added to a container, but to the whole space instead. This is the case for all aspects added at a global interception point. In addition to that, local aspects, which are not defined on one specific container but on all existing and newly created containers, are also handled by this class. The implementation of this class is simple, since it is able to reuse the aspect manager, having it enveloped in the class (`private static AspectManager aspectManager`). The biggest difference between the aspect manager and the global aspect manager, is the number of instances within `MozartSpaces`. Whereas there might be multiple instances of the aspect manager (according to the number of containers), there is always only one instance of the global aspect manager. Therefore, all defined methods in the `GlobalAspectManager` class are static. Aspects can be added/removed by calling `addAspect` respectively the `removeAspect` method, which is execution is simply delegated to the aspect manager. Since the global aspect manager is a shared resource it needs to be synchronized. This is done by using the `java.util.concurrent` package, which provides a `ReadWriteLock` class³. Using this class it is possible to synchronize threads as shown in listing 3.7. The `ReadWriteLock` (line 2) exposed lock and unlock mechanisms used in line 5,7 and line 11 to achieve exclusive access to the aspect manager.

```

1
2     private static ReadWriteLock rwlock = new ReentrantReadWriteLock();

```

³<http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/locks/ReadWriteLock.html>

```

3
4 public static void removeAspect(IPoint p, IAspect aspect) {
5     rwlock.writeLock().lock();
6     aspectManager.removeAspect(p, aspect);
7     rwlock.writeLock().unlock();
8 }
9
10 public static void addAspect(IPoint p, IAspect aspect) {
11     rwlock.writeLock().lock();
12     aspectManager.addAspect(p, aspect);
13     rwlock.writeLock().unlock();
14 }

```

Listing 3.7: Global Aspect Manager Thread Synchronization

The next section deals with the whole process of aspect execution.

3.6.5. Aspect Execution Process

This section will explain in detail, how added aspects are called within the Java implementation MozartSpaces. The sequence diagram in figure 3.6 gives a general overview over the systems mechanisms to trigger the different types of aspects.

As shown in the overview the order of aspect triggering is as followed:

1. Global Pre Aspects
2. Local Pre Aspects
3. Local Post Aspects
4. Global Post Aspects

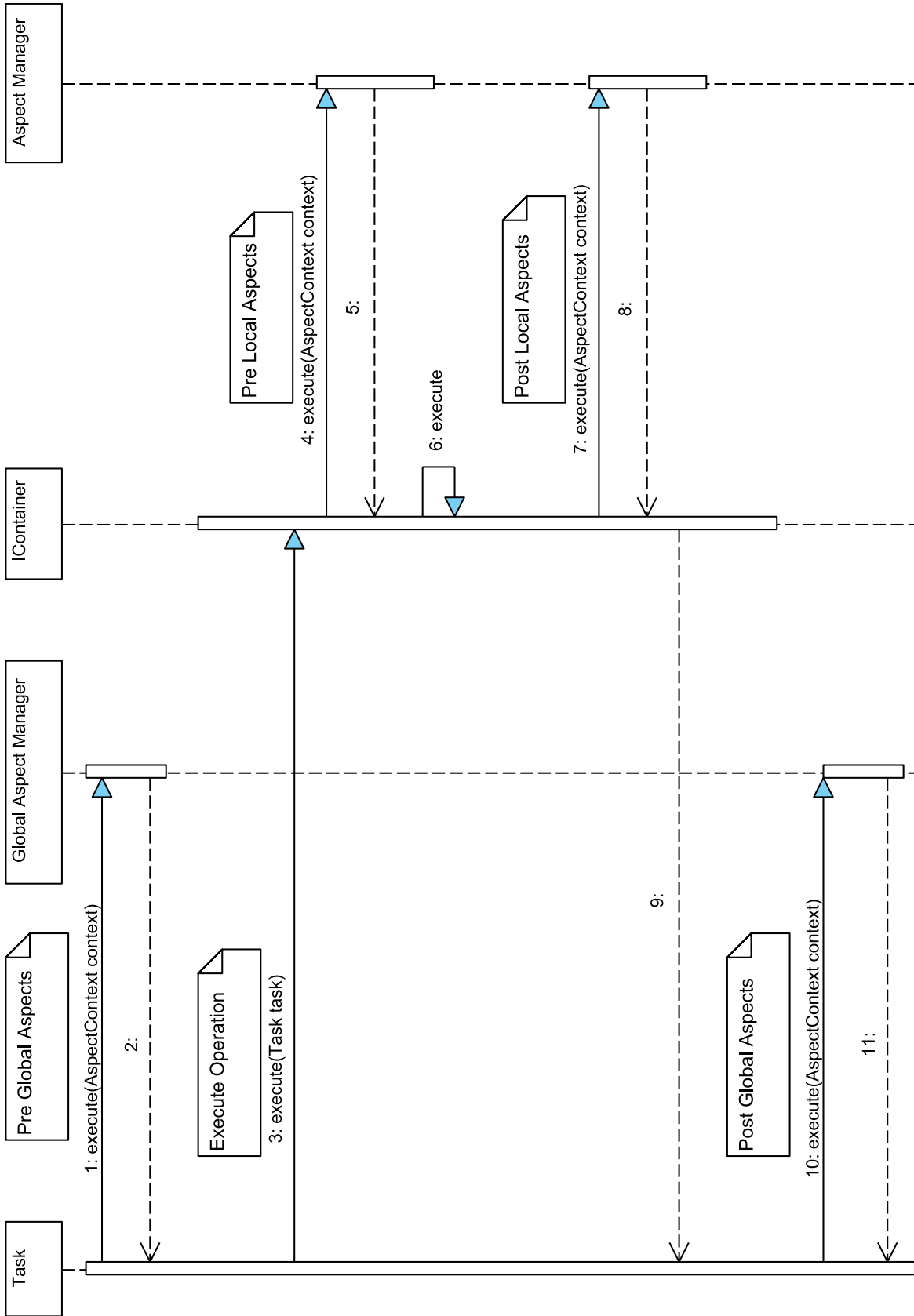


Figure 3.6.: Aspect Triggering Overview

This order is a result of the layer architecture, shortly described in 1.2.3. A detailed description of the MozartSpace architecture can be found in [18]. Starting at the triggering of the global aspects: there are to different kinds of aspects, XVSM refers to as global aspects. On the one hand, the aspects which are not bound to one container (e.g.: Aspects on transaction create, Container create) and on the other hand aspects, which are only triggered by operations on one specific container, but are automatically added to every container in the space. As described in section 1.2.3 every operation within MozartSpaces is represented by runnable task objects. The following classes are responsible for triggering global aspects depending on the according operation:

- AspectTask class
- ContainerTask class
- OperationTask class
- ShutdownTask class
- TransactionTask class

The mechanism of triggering the global aspects is nearly the same in every class. At first an instance of the aspect context class is created and the attributes are set, according to table 3.5. After that, the global aspect manager is called with this context in order to trigger the aspects. The process is shown in listing 3.8. The only differences between those classes, is the information set in the created aspect context.

```
1 AspectContext acontext = new AspectContext();
2
3 try {
4     GlobalAspectManager.execute(acontext);
5
6 } catch (AspectNotOkException e) {
7     this.setResult(e);
8     return;
9 } catch (AspectRescheduleException e) {
10    EventProcessingPool.getInstance().execute(this);
11    return;
12 } catch (AspectSkipException e) {
13    skip = true;
14 }
```

Listing 3.8: Global Pre Aspect Triggering

In order to react on an aspect's return value, the according exceptions have to be caught. In the case of a not ok return value, the result of this operation is simply set to this exception. Otherwise, if the task has to be rescheduled, the task is given to the event processing pool and finishes its execution by returning. The boolean skip, which indicates that an aspect wants to skip the operation is used to decide whether an operation should be executed, after the triggering of the pre aspects or not. Therefore execution will only take place if the boolean skip equals false.

Triggering global post aspect is also done by the classes mentioned above and it is also based on the same mechanisms. The only difference here again is the information set in the aspect context. In addition to that, exceptions thrown by the aspect are handled differently. Since a global post aspect is never allowed to have aspect return values, according to table 3.4, an exception is set as result for the operation as shown in listing 3.9.

```
1 try {
2     GlobalAspectManager.execute(acontext);
3 } catch (Exception e) {
4     this.setResult(new FatalException("Aspect result not allowed here. " + e
5         ));
6     return;
7 }
```

Listing 3.9: Global Post Aspect Triggering

In addition to global aspects, MozartSpaces also supports the addition and triggering of local aspects. As shown in the overview in figure 3.6 the responsibility of triggering local aspects is within the IContainer interface. Considering the layered architecture of a container as described in section 1.2.3 the transaction layer is capable of triggering aspects. Aspect related exceptions and method calls are simply forwarded from the blocking to the transaction layer. As the global aspect manager the transaction layer has an instance of the AspectManager class. The difference is, that every container has its own instance and since the blocking layer only allows one thread to enter, there is no need for thread synchronization within the aspect manager. The aspect relevant specified methods in the interface are shown in listing 3.10.

```
1 List<Entry> read(Transaction tx, List<Selector> selectors, int retrycount,
2     Properties aspectContext) throws NoSuchCoordinationTypeException,
3     CountNotMetException, InvalidTransactionException,
4     TransactionLockException, AspectRescheduleException,
5     AspectNotOkException;
6
7 List<Entry> take(boolean isDelete, Transaction tx,
```

```

8     List<Selector> selectors, int retrycount, Properties aspectContext)
9     throws NoSuchCoordinationTypeException, CountNotMetException,
10    InvalidTransactionException, TransactionLockException,
11    AspectRescheduleException, AspectNotOkException;
12
13
14 void write(List<Entry> entries, Transaction tx, int retrycount,
15           Properties aspectContext) throws ContainerFullExcpetion,
16           NoSuchCoordinationTypeException, InvalidTransactionException,
17           TransactionLockException, AspectRescheduleException,
18           AspectNotOkException;
19
20 List<Entry> shift(List<Entry> entries, Transaction tx,
21                 Properties aspectContext) throws NoSuchCoordinationTypeException,
22                 InvalidTransactionException, TransactionLockException,
23                 AspectRescheduleException, AspectNotOkException;

```

Listing 3.10: ITransactionLayer Interface

Since a destroy operation is mapped to a take operation, it is not specified in the interface. The exact meaning of all parameters, which are not aspect related and the interface is explained in detail in [18]. As far as aspects are concerned, the only aspect related objects are the aspectContext parameter and the possibility to throw according aspect exceptions. Since the AspectSkipException is handled by the transaction layer, there is no need to pass that exception to the layer above. The other two exceptions involve rescheduling and throwing exceptions and therefore they can not be processed within the context of the container and have to be rethrown. The blocking layer itself just increases the retry count `task.setRetrycount(task.getRetrycount()+1)` of the task and rethrows all exceptions. In the following, the task will either simply reschedule itself or initiate the rollback of the transaction, depending on whether a reschedule or a not ok exception was thrown.

The last thing necessary to know, is how aspects are added respectively removed by using the core API (CAPI). The aspect related methods in the ICapi interface are shown in listing 3.11.

```

1 void addAspect(ContainerRef cref, List<LocalIPoint> p, LocalAspect aspect)
2     throws XCoreException;
3
4 void addAspect(URI site, List<IPoint> p, GlobalAspect aspect)
5     throws XCoreException;
6

```

```

7 void removeAspect(ContainerRef cref, List<LocalIPoint> p, LocalAspect
   aspect)
8     throws XCoreException;
9
10 void removeAspect(Uri site, List<IPoint> p, GlobalAspect aspect)
11     throws XCoreException;
12
13 void setAspectContext(Properties aspectContext) throws XCoreException;

```

Listing 3.11: ICapi Aspect Methods

The first `addAspect` method is used, when adding local aspects. Therefore the reference to the container has to be passed, where the aspect should be added. In addition to that, a list of `LocalIPoints` has to be specified, according to when the aspect should be triggered. The last parameter is an instance of the aspect itself. Since local aspects can also be added to every existing container as well as to newly created ones, as described in section 3.2, the `MozartSpace ICapi` allows the passed container reference to be null. This will lead to the behavior of adding the aspect to every container. The removal of an aspect is done by calling the `removeAspect` method. This takes exactly the same parameters as the complementary method, with the same semantics. Since this method also takes a list of `LocalIPoints` it is possible to only remove specific `IPoints` of an aspect. As a result, the aspect is no longer triggered by these `IPoints`.

The only difference between those methods, and the related ones, for adding or removing global aspects, is that a Java `URI` object is used, instead of a container reference, which is a result of the identification of a `MozartSpace` core. In order to address and identify a running core a `URI` object is used. Since global aspects are not added to a container, but to the whole space instead, the `URI` is passed to the according method in order to specify the `MozartSpace` core. The identification and handling of multiple `MozartSpace` cores is handled in [18].

The last method `setAspectContext` allows users to set a context, which is passed to every triggered aspect. As described in 3.6.3 an instance of a simple Java property class⁴ is used to pass the according values to the aspects.

The next section will show on the basis of a simple example, how aspects are used within `MozartSpaces`.

⁴<http://java.sun.com/j2se/1.5.0/docs/api/java/util/Properties.html>

3.7. An Approach for Implementing Access Control by means of Aspects

This section describes an approach for implementing access control within MozartSpaces. Therefore the described mechanism of aspects is used. The main goals of this access control are as follows:

- Access control based on a password
- Preventing any operation on containers without the knowledge of the password
- Prevent addition of new aspects before the security aspect
- Prevent removal of the existing aspect

In order to achieve this, the approach is to develop an aspect, which is capable of the goals listed above and which can be added to every container. The password should be created when the aspect is added to the container and is passed to the aspect using the aspect context. In this version of MozartSpaces it is not possible to specify the order of aspects on adding. In order to grant, that the security aspect is the first to execute the following steps are necessary:

1. Create a container under a transaction
2. Add the security aspect to that container
3. Commit the transaction

When applying these three steps it is impossible that someone else added an aspect before the security aspect.

In order to start the implementation of this aspect a class `AccessControl` is created, extending `LocalAspect` as shown in listing 3.12.

In order to set the password the according constructor is created, taking a string as a parameter, calculating and storing a hash code. The logic that has to be implemented here is the check of the hash code. Therefore the passed password is retrieved out of the user transmitted context and is checked against the stored one. If they are different the access control aspect throws an according not ok exception or returns otherwise. The whole implementation is shown in listing 3.13.

```

1 public class AccessControl extends LocalAspect {
2
3     @Override
4     public void execute(AspectContext c) throws AspectNotOkException,
5         AspectRescheduleException, AspectSkipException {
6
7         // Access Control Logic
8     }
9 }

```

Listing 3.12: Access Control Stub

```

1 public class AccessControl extends LocalAspect {
2
3     private int hash;
4
5     public AccessControl(String password) {
6         hash = password.hashCode();
7     }
8
9     @Override
10    public void execute(AspectContext c) throws AspectNotOkException,
11        AspectRescheduleException, AspectSkipException {
12        String password = c.getAspectsContext().getProperty("password");
13        if (password == null || hash != password.hashCode()) {
14            throw new AspectNotOkException("Access denied");
15        }
16    }
17 }

```

Listing 3.13: Access Control Aspect

In order to use the access control, the aspect just has to be added on a container. Since the aspect is only triggered on the given LocalIPoints in the addAspect method, it is easy to restrict only a few operations. Anyway, in this example the access control is used to prevent not authorized users from executing any operation on a container. Therefore the access control is added to the container as described in listing 3.14.

```

1 List<LocalIPoint> ipoints = new ArrayList<LocalIPoint>();
2 ipoints.add(LocalIPoint.PreAddAspect);
3 ipoints.add(LocalIPoint.PreDestroy);
4 ipoints.add(LocalIPoint.PreRead);
5 ipoints.add(LocalIPoint.PreRemoveAspect);
6 ipoints.add(LocalIPoint.PreShift);

```

```
7 ipoints.add(LocalIPoint.PreTake);  
8 ipoints.add(LocalIPoint.PreWrite);  
9 capi.addAspect(cref, ipoints, new AccessControl("secret_password"))
```

Listing 3.14: Adding Access Control

Every operation performed will end up in an exception, unless the password is set using the `setAspectContext` method in the CAPI.

As shown in this simple and very short example, the usage of aspects in combination with their return values, is a very powerful mechanism. The next chapter describes how aspects are also used to implement the mechanism of notifications within MozartSpaces. At first the mechanism and its semantics are explained in general, before focusing on the implementation issues in detail.

Chapter 4.

Semantics and Implementation of Notifications

This chapter introduces another mechanism, which XVSM supports. The first section in this chapter explains the general purpose and advantages of notifications. The following section describes the semantics of notifications within XVSM. After that the implementation within MozartSpaces is explained in detail, followed by the last section, which deals with future works and implementation issues of MozartSpaces.

4.1. Notifications in General

Necessary to understand the needs and the purpose of notifications in general, the following use case is considered: Having a peer A and another peer B. Peer A wants to be informed whenever a certain event occurs at peer B. In order to achieve this, one of the following mechanisms can be used:

- Polling
- Notifications

The **polling** mechanism is shown in figure 4.1.

In order to check if a certain event has occurred at peer B, peer A just asks periodically for new information. This approach is also referred to as an active one, since peer A has to actively query for new information.

The second approach, leads to the use of **notifications**.

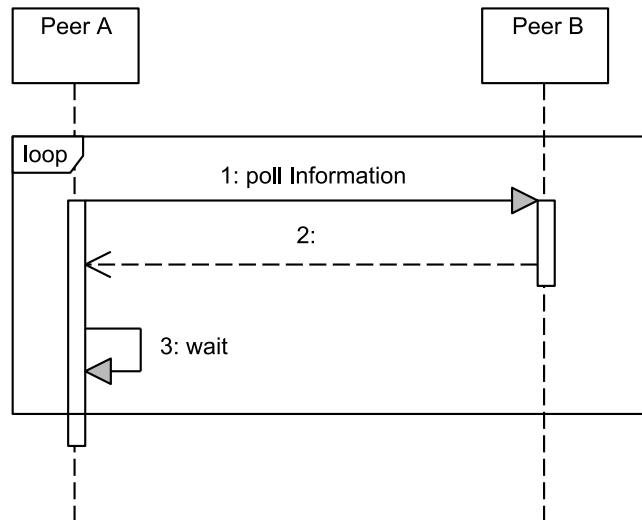


Figure 4.1.: Polling Mechanism

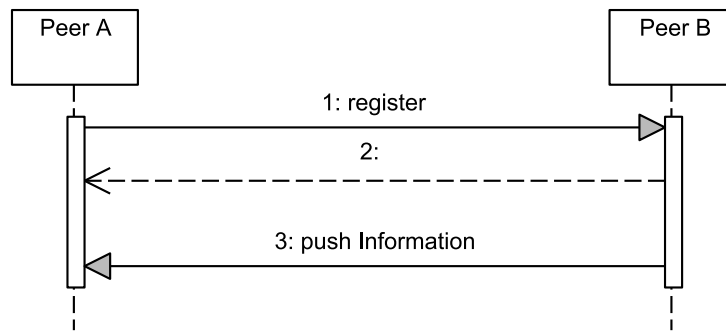


Figure 4.2.: Notification Mechanism

“Notification is a very useful feature for internet applications. Users can recognize that something happened passively without checking actively” [3].

As shown in figure 4.1 peer A just has to register itself at peer B. This registration tells peer B to inform peer A when new information is available. Additionally it could even pass the information to peer B.

Comparing the different approaches, the notifications have several advantages over the polling mechanism. The most important ones are listed below:

- Decreasing the central processing unit (CPU) load
- Decreasing network load
- Increase performance

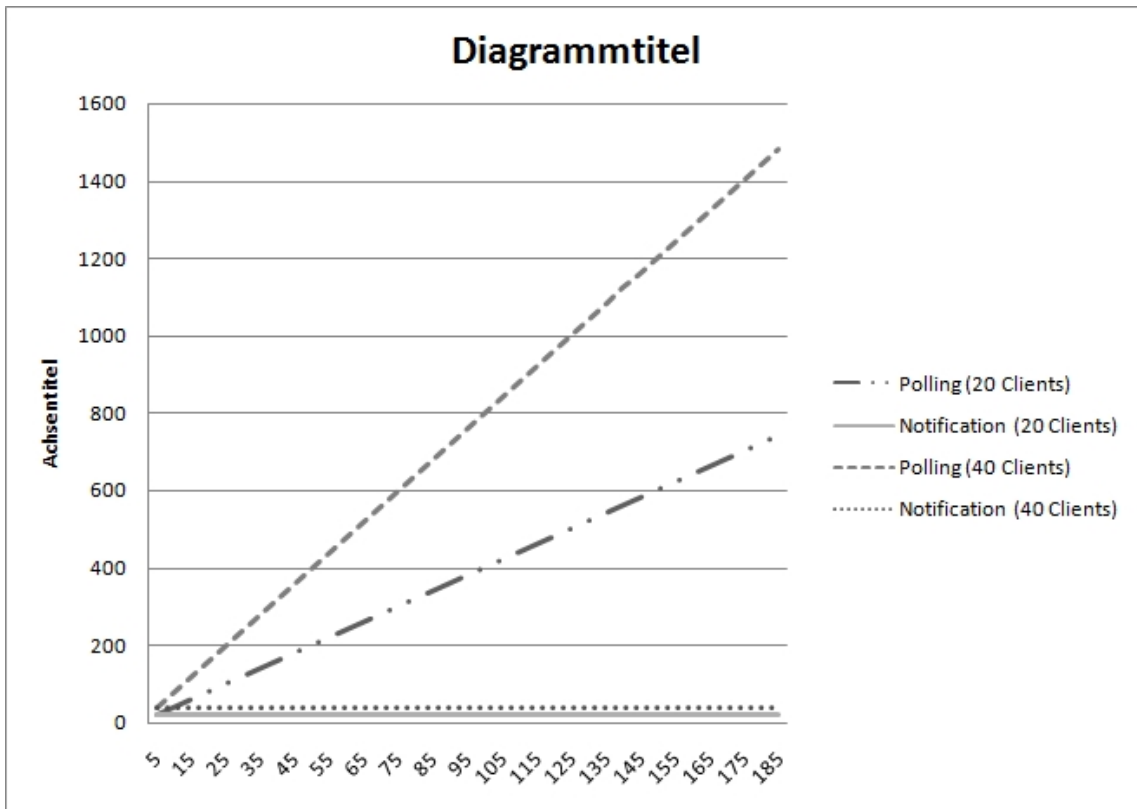


Figure 4.3.: Number of Requests (Polling & Notification)

Decreasing the load of the CPU is an advantage, which is applied on both sides. On the side of the requester (peer A), there is no need for processing the query and on the other side (peer B) the number of requests and their processing is decreased. Decreasing the number of requests has a decrease of **network load** as a result. Since the network is only used during registration, respectively sending new information the traffic is reduced to a minimum. As a consequence of the mentioned advantages, the **performance increases**. This can be illustrated by the following simple example. Considering an amount of α clients want to be informed, when new data is available. All of them poll the information provider every λ seconds. Assuming that there is no new information within the next Δ seconds, the total number of request Φ is calculated as follows: $\Phi = \frac{\alpha}{\lambda} * \Delta$. Using notifications instead, the number of requests is reduced to the number of notification registrations. Since each client only has to register once, the number of processed requests is equal to the number of clients α . Figure 4.3 shows the number of requests within 180 seconds for 20, respectively 40 clients.

In addition to the polling strategy XVSM also supports the mechanism of notifications. Therefore the next section explains the semantics of notification within XVSM in detail.

4.2. Semantics of Notifications

The design of the notification mechanism in XVSM is to support different semantics. The so called *flavors of notifications* differ in their behavior as well as in the guarantees they give about the state of the space. As described in the JavaSpaces Service Specification ¹ there is only one kind of notification. This notification type only supports the notification on new written entries having a template base selection.

The most important thing to know is that the XVSM protocol itself, does not contain any notification related elements. This is the result of the idea to support many different semantics, rather than one, which is limited to the protocol. In order to achieve this important goal, the protocol forces implementations to rely on existing mechanism. As explained in chapter 3 the XVSM protocol supports aspects. XVSM recommends implementations to use aspects in order to achieve the support of many different flavors of notifications.

Independent of the notification's flavor, some mechanisms should always remain the same. In general notifications in XVSM are registered at a certain container with the purpose of monitoring operations related to it. Notifications are able to choose the operations they want to monitor by having one or more of the following targets:

- Read
- Take
- Destroy
- Write
- Shift

In addition to these targets every interception point can be used to trigger a notification. This version of MozartSpaces only supports the targets listed above. Since XVSM does not limit the semantics of notifications, but instead enables them to be as powerful as necessary for a specific domain, there are no general suggestions for the firing of notifications.

¹http://www.sun.com/software/jini/specs/js2_0.pdf

Since XVSM realizes the mechanism of notification by providing fundamental mechanism to realize any kind of notification, they can be seen as applications rather than integrated features. The main differences concerning the semantics of notifications are as follows:

- Trigger of firing
- Visibility under transactions
- Selectivity
- Return values
- Guaranties

The issue of the firing trigger is the same as with aspects (described in section 3.2). Differences might exist between the firing reason. Should a notification fire because of an specific operation or on the related consequences.

The visibility under transactions also plays an import role. Notification might fire within transactions or only after a transaction has been committed.

In addition, notifications might fire because of the operations performed under a transaction, rather than on the real consequences. For example, considering an entry to be written and removed under the same transaction. Notification might fire, because of the executed operations (write, destroy) or does not fire at all, because the content of the container has not been changed.

Selectivity also plays an important role. It allows users to specify additional constraints (e.g. entry with a special value), when they want to be informed. Some notifications might not support this, whereas others do.

Flavors might also differ in the information, they give back to the client. They might return a specific entry or nothing at all, simply informing the client, that something changed without giving it any information about the change itself.

The last mentioned possible difference between the flavors of notifications might be the guaranties they provide, when firing. Notifications could grant, that an entry is still in the space or do not grant anything at all.

In addition to the mentioned possible differences, there exist many other flavors of notifications. The enumeration above is not complete since every semantic of notifications can be implemented using the mechanisms of aspects.

The next section deals with the issue of the implementation within MozartSpaces. Therefore the realized notification flavor is described in detail, as well as its implementation.

Notification Target/Operation	Read	Take	Destroy	Write	Shift
Read	X				
Take		X			
Destroy			X		
Write				X	
Shift					X

Table 4.1.: Notification Firing

4.3. MozartSpaces Notification Implementation

MozartSpaces supports the notification mechanism introduced in section 4.2. This section will explain the semantics of the implemented notification flavor in MozartSpaces. After that the implementation itself is explained in section 4.3.2.

4.3.1. Semantic of the implemented Notification Flavor

For simplification the term notification is used for the flavor of notification implemented in MozartSpaces in the following.

The main purpose of the implemented notification is to inform what has happened on a specified container. The following notification targets are supported: read, take, destroy, write and shift. This notification also has the following characteristics:

- No selection
- No guarantees are given after firing
- Able to return the triggering data
- Firing only on successfully executed operations

In addition to that, the *lease* of a notification should also be parameterized. This means, that a notification should be able to only fire for a specific amount of times, unregistering itself afterward.

In order to support an easy semantics of notifications in the first place, MozartSpaces chose to use the firing semantics as shown in table 4.1.

Write: Notifications with the target write are supposed to fire, whenever an entry was written to the container. It is not guaranteed that the written entry is still part of that container

after the notification has fired. If it is supposed to return entries, the written ones will be returned.

Read: When read is set as a target the notification will fire, every time a read operation is performed on the container and will not fire if someone performs a take operation. It returns the read entries if specified.

Take: Notifications, which have take as a target, are supposed to fire each time someone took an entry out of the container. If specified, it will return the taken entries.

Shift: Setting the shift target leads the notification to fire each time an entry was written using the shift operation. It returns the written entries as well as the removed ones, in the course of the operation destroys entries.

Destroy: Setting the target destroy will lead the notification to fire whenever an entry was destroyed by the destroy operation. It will not fire when entries were destroyed because of a take or a shift operation. The destroyed entries are returned, if configured.

As already described in section 3.2, it is the simpler approach to fire because of the executed operations rather than on their effects. Since the implementation of aspects also follows this approach, it is also realized for the firing of notifications. This is done in order to achieve a homogeneous semantics through the whole implementation of MozartSpaces.

As far as the visibility under transactions is concerned, this flavor of notifications is not able to monitor changes within a transaction. The notification only fires after the commit of a transaction, but for all operations performed under that transaction. It fires because of the executed operations and not on the transactions effects on the container.

E.g.: Transaction (tx) writes 5 entries (e1, ..., e5) into a container, monitored by a notification (n) with the write target set. The following operations are executed:

1. Write entries e1,...,e5 under transaction tx
2. Destroy entries e1, ..., e3 under transaction tx
3. Commit transaction tx

On the commitment the notification is supposed to fire five times, because five write operations were executed under the committed transaction, even if the overall effect of the transaction is the addition of entry e4 and e5.

One of the last important things, necessary to specify the semantics of the implemented notification is, that it does not provide any selectivity in MozartSpaces version 1.0 yet.

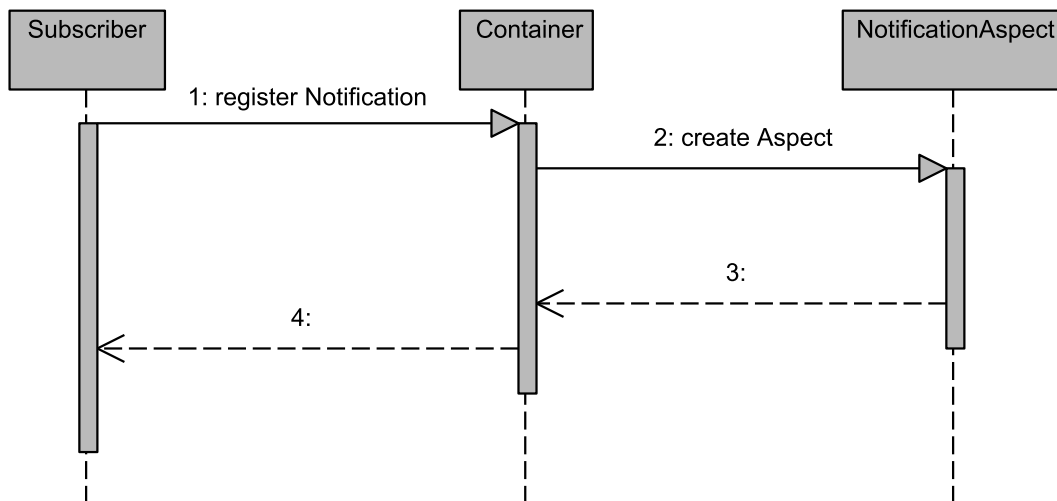


Figure 4.4.: Notification Overview

The next section deals with the implementation itself. It also describes the whole process from the registration to the firing and the used aspects in detail.

4.3.2. Notification Implementation Details

As suggested, MozartSpaces uses aspects to realize notifications. The general idea is shown in the sequence diagram 4.4.

Whenever a subscriber registers a notification at a certain container, a notification aspect is created. The purpose of this aspect is to copy entries of interest into a specified container. The subscriber site only has to perform a blocking take operation on the newly created container in order to receive these entries.

In detail, a subscriber in MozartSpaces is represented by implementing the NotificationListener interface. This interface only prescribes one method named handleNotification, which takes a NotificationContext as parameter. This context class has the attributes shown in listing 4.1. It also implements the according setter and getter methods.

This context is given back to the client, whenever a notification is fired. The container reference represents the container where the notification was fired, whereas the according target is also specified. In the case that entries were added or removed, they are set in the entries array. According to the semantics defined in section 4.3.1 shift returns the written entries, which are stored in the entries array. In addition to that MozartSpaces also returns destroyed

```

1 public class NotificationContext {
2
3     private ContainerRef cref;
4     private NotificationTarget target;
5     private Entry[] entries;
6     private Entry[] shifted;

```

Listing 4.1: Notification Context

Notification Target	Interception Point
Read	Post Read
Write	Post Write
Take	Post Take
Destroy	Post Destroy
Shift	Post Shift

Table 4.2.: Mapping: Notification Target to Join Points

entries. These entries are set in the shifted array.

In order to create a notification on a container, the ICapi interface provides the method shown in listing 4.2.

```

1 void createNotification(ContainerRef cref, int firingCount,
2     NotificationTarget target, boolean returnEntries,
3     NotificationListener listener) throws XCoreException;

```

Listing 4.2: Create Notification Interface

This method takes the container reference of the container where the notification should be created. The firingCount is used to specify the maximum number a notification should fire using -1 for infinite. An notification target, which can be one of the targets mentioned in section 4.2 is also passed. The notification listener, which is informed in the case of firing is also specified. The boolean returnEntries enables to decide, whether the notification should return entries or not.

MozartSpaces uses an aspect to copy entries to the created container according to their notification targets. These aspects are added to the local interception points, depending on the notification target. Table 4.2 shows the mapping between notification targets and the interception points, where the aspect is added.

The implementation of the according methods and the constructors is shown in listing 4.3.

The constructor of a notification aspect takes the container reference of the container, were

```

1 public NotificationAspect(ContainerRef notifyContainer, int firingCount) {
2     this.firingCount = firingCount;
3     this.notifyContainer = notifyContainer;
4 }
5
6 @Override
7 public void postWrite(ContainerRef cref, Transaction tx,
8     List<Entry> entries, Properties context) {
9     this.entriesToNotifyContainer(tx, entries);
10    if (this.checkNotificationExpired(cref)) {
11        this.removeAspect(cref, LocalIPoint.PostWrite);
12    }
13 }
14
15 @Override
16 public void postRead(ContainerRef cref, Transaction tx,
17     List<Entry> entries, List<Selector> selectors, Properties context) {
18    this.entriesToNotifyContainer(tx, entries);
19    if (this.checkNotificationExpired(cref)) {
20        this.removeAspect(cref, LocalIPoint.PostRead);
21    }
22 }
23
24 @Override
25 public void postDestroy(ContainerRef cref, Transaction tx,
26     List<Selector> selectors, List<Entry> deleted, Properties context) {
27    this.entriesToNotifyContainer(tx, deleted);
28    if (this.checkNotificationExpired(cref)) {
29
30        this.removeAspect(cref, LocalIPoint.PostDestroy);
31    }
32 }
33
34 @Override
35 public void postTake(ContainerRef cref, Transaction tx,
36     List<Selector> selectors, List<Entry> taken, Properties context) {
37    this.entriesToNotifyContainer(tx, taken);
38    if (this.checkNotificationExpired(cref)) {
39        this.removeAspect(cref, LocalIPoint.PostTake);
40    }
41 }
42
43 @Override
44 public void postShift(ContainerRef cref, Transaction tx,
45     List<Entry> entries, List<Entry> shifted, Properties context) {
46    this.entriesToNotifyContainer(tx, entries);
47    if (this.checkNotificationExpired(cref)) {
48        this.removeAspect(cref, LocalIPoint.PostShift);
49    }
50 }

```

Listing 4.3: Notification Aspect

it should copy the according entries. In addition to that, the number maximal fire count is passed as an argument.

Since the same class is used for all kinds of notification targets, it overwrites all the according post methods of the extended local aspect class. The process in every method is nearly the same. All methods use the private utility method `entriesToNotifyContainer` in order to copy the entries given into the notifications container. After that it is checked, whether the notification has already reached its maximum firing count or not. If it has reached its limit, it will unregister itself by using the private method `removeAspect`.

As already described, the CAPI is responsible for executing the take operation on the container and calling the `handleNotification` method in the according notification listener. These operations are totally hidden from the client, so that it only has to implement the listener interface and create a notification passing itself as the listener.

The last chapter of this work before the conclusion deals with an perspective of future work. Therefore possible use cases and further development steps are mentioned.

Chapter 5.

Future Work and Conclusion

This chapter deals with future work and gives a conclusion of this work in section 5.2.

5.1. Future Work

Since the XVSM protocol itself provides highly extensible mechanisms such as aspects and custom coordinators, which are described in [18], there are many perspectives of future work.

In version 1.0 the supported communication protocols related to the MozartSpace implementation are Java binary streaming and xml, as described in [18]. In order to be able to communicate with other technologies, different bindings should be developed. E.g.: bindings for messages services like jms¹ and interfaces for representational state transfer (REST²) or web services. In order to be able to replace JavaSpaces within applications, bindings for MozartSpaces are developed as well.

Another important goal of the further development is the implementation of automatically discovery between different MozartSpace cores. Within a network this can be simply implemented with the use of multicasts. In addition to that, the client could also discover running cores automatically, so that the application on the layer above does not even know to which core it is connected. As a result of this approach, highly availability can be granted by implementing failover mechanisms on the client side. This would mean, that a client automatically contacts another core in the case of failure of the previous one. This approach would grant high availability as well as total transparency on the client.

¹<http://java.sun.com/products/jms/>

²http://en.wikipedia.org/wiki/Representational_State_Transfer

Developing domain specific higher level coordinators is also an important perspective. Auction coordinators, which only allow writing higher bids in a container can be implemented. Coordinators, which provide voting mechanisms can also be realized in further development.

The last section of this work, will summarize the gained experience and gives a conclusion of the achieved goals related to a next generation middleware.

5.2. Conclusion

The purpose of this thesis was to describe and implement mechanisms, which support the needs of a next generation middleware. XVSM fulfills the space-based computing paradigm, which is described in [20] by running a virtual server, based on a peer-to-peer network. In addition to that, the XVSM protocol provides powerful mechanisms in order to extend the functionality, according to needs of a specific business domain. The focus of this thesis was on the following facets:

- Timeouts
- Aspects
- Notifications

The mechanism of **timeouts** allows to specify a certain amount of time in which an operation or transactions should be executed. This also leads to a very simple but effective way of handling deadlocks as described in chapter 2.

Aspects are possibly the most important mechanism within the XVSM protocol. Using aspects provides to most powerful mechanism by giving developers features of aspect oriented programming, such as component reusability or modularity. Since every available information is passed to an aspect it can be as complex as needed. Instead of just providing the information of the core itself, XVSM also allows aspects to exchange information with the application using a user defined context. This mechanisms allows developers to implement access control by only writing a few lines of code as shown in section 3.7.

In order to avoid unnecessary exchange of data, XVSM uses the mechanism of **notifications**. Instead of sticking to one specific kind of notification in the protocol, XVSM enables developers to implement their own notification mechanism. Many different flavors of notifications are supported, because the XVSM protocol does not contain any specification of

notifications. Instead it uses to the existing mechanism of aspects, which therefore provides this great amount of possibilities of notification implementations.

The XVSM implementation MozartSpaces ³ already proved that it is more than a prototype. It was already successfully used by 80 students for two semesters in the lecture of A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn ⁴ and is enjoying its increasing open source community.

³www.mozartspaces.org

⁴<http://www.complang.tuwien.ac.at/eva/Teaching/SBC/sbcIndex.html>

Bibliography

- [1] Md. Nawab Yousuf Ali and Mohammad Zakir Hossain Sarker. An algorithm for avoiding deadlock. *9th International Multitopic Conference, IEEE INMIC 2005*, pages 1–6, Dec. 2005.
- [2] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: employing hardware migration to specialize cmp cores on-the-fly. *SIGARCH Comput. Archit. News*, 34(5):283–292, 2006.
- [3] Chi-Huang Chiu, Ruey-Shyang Wu, Chi-lo Tut, Hsien-Tang Lin, and Shyan-Ming Yuan. Next generation notification system integrating instant messengers and web service. In *ICCIT '07: Proceedings of the 2007 International Conference on Convergence Information Technology*, pages 1781–1786, Washington, DC, USA, 2007. IEEE Computer Society.
- [4] Siobhàn Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design*. Addison-Wesley Professional, 2005.
- [5] Severin Ecker. Communication protocols in xvsm- design and implementation. Master's thesis, Vienna University of Technologie Institute of Computerlanguages, August 2007.
- [6] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
- [7] V.D. Gligor and S.H. Shattuck. On deadlock detection in distributed systems. *Software Engineering, IEEE Transactions on*, SE-6(5):435–440, Sept. 1980.
- [8] Ram Goverdhana and M.E. Fayad. Any transaction stable design pattern. *Information Reuse and Integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on*, pages 54–59, 8-10 Nov. 2004.
- [9] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

- [10] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
- [11] Markus Karolus. Design and implementation of xcospaces, the .net reference implementation of xvsm (in preparation). Master's thesis, Vienna University of Technologie Institute of Computer Languages, 2008.
- [12] E. Kühn, R. Mordinyi, and C. Schreiber. An extensible space-based coordination approach for modeling complex patterns in large systems. *3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Special Track on Formal Methods for Analysing and Verifying Very Large Systems*, 2008.
- [13] 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.
- [14] D. Matic, D. Butorac, and H. Kegalj. Data access architecture in object oriented applications using design patterns. *Electrotechnical Conference, 2004. MELECON 2004. Proceedings of the 12th IEEE Mediterranean*, 2:595–598 Vol.2, 12-15 May 2004.
- [15] S.K. Miller. Aspect-oriented programming takes aim at software complexity. *Computer*, 34(4):18–21, Apr 2001.
- [16] David Reilly. Simple handling of network timeouts. Website, October 1999. Available online at <http://java.sun.com/developer/technicalArticles/Networking/timeouts/>; visited on March 28th 2008.
- [17] Thomas Scheller. Design and implementation of xcospaces, the .net reference implementation of xvsm - core architecture and aspects (in preparation). Master's thesis, Technical University Vienna, Insitute of Computer Languages, 2008.
- [18] Christian Schreiber. Design and implementation of mozartspaces, the java reference implementation of xvsm - custom coordinators, transactions and xml protocol. Master's thesis, Vienna University of Technologie Institute of Computer Languages, 2008.
- [19] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *OOP-SLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 481–497, New York, NY, USA, 2006. ACM.

- [20] Dezheng Xu, Xiaoying Bai, and Guilan Dai. A tuple-space-based coordination architecture for test agents in the mast framework. *Service-Oriented System Engineering, 2006. SOSE '06. Second IEEE International Workshop*, pages 57–66, Oct. 2006.

Appendix A.

MozartSpaces API

shutdown(site, clearSpace)

description	Shutsdown a MozartSpaces core
parameter	site (URI): the identifier of the core clearSpace (boolean): specifies if all entries should be deleted

clearSpace(site)

description	Clears all entries
parameter	site (URI): the identifier of the core

lookupContainer(tx, site, containerName)

description	Lookup a specific named container
parameter	tx (Transaction): the transaction of the operation site (URI): the identifier of the core containerName (String): the name of the container

lookupMetaContainer(tx, cref)

description	Lookup a meta container
parameter	tx (Transaction): the transaction of the operation cref (ContainerRef): the container for which to lookup the meta container

createContainer(tx, site, name, size, coordinators)

description	Creates a new container
parameter	tx (Transaction): the transaction of the operation site (URI): the identifier of the core name (String): the name of the container size (int): the size of the new container coordinators (ICoordinators[]): the supported coordinators

destroyContainer(tx, cref)

description	Destroys a container
parameter	tx (Transaction): the transaction of the operation cref (ContainerRef): the container on which the operation is executed

read(cref, timeout, tx, selectors)

description	Read entries from a container
parameter	cref (ContainerRef): the container on which the operation is executed timeout (long): the specific timeout of the operation tx (Transaction): the transaction of the operation selectors (Selector[]): the selectors of the operation

take(cref, timeout, tx, selectors)

description	Take entries from a container
parameter	cref (ContainerRef): the container on which the operation is executed timeout (long): the specific timeout of the operation tx (Transaction): the transaction of the operation selectors (Selector[]): the selectors of the operation

destroy(cref, timeout, tx, selectors)

description	Destroy entries from a container
parameter	cref (ContainerRef): the container on which the operation is executed timeout (long): the specific timeout of the operation tx (Transaction): the transaction of the operation selectors (Selector[]): the selectors of the operation

write(cref, timeout, tx, entries)

description	Write entries in a container
parameter	cref (ContainerRef) : the container on which the operation is executed timeout (long) : the specific timeout of the operation tx (Transaction) : the transaction of the operation entries (Entry[]) : the entries to write

shift(cref, timeout, tx, entries)

description	Shift entries in a container
parameter	cref (ContainerRef) : the container on which the operation is executed timeout (long) : the specific timeout of the operation tx (Transaction) : the transaction of the operation entries (Entry[]) : the entries to write

createNotification(cref, firingCount, target, returnEntries, listener)

description	Creates a notification on a container
parameter	cref (ContainerRef) : the container on which the operation is executed firingCount (int) : the maximum number of notification firing returnEntries (boolean) : if entries should be returned or not listener (NotificationListener) : the notification listener

createTransaction(site, timeout)

description	Creates a transaction
parameter	site (URI) : the identifier of the core timeout (long) : the specific timeout of the transaction

commitTransaction(tx)

description	Commits a transaction
parameter	tx (Transaction) : the transaction to commit

rollbackTransaction(tx)

description	Rolls back a transaction
parameter	tx (Transaction) : the transaction to rollback

addAspect(cref, ipoints, aspect)

description	Adds an aspect to a container
parameter	cref (ContainerRef) : the container on which the operation is executed ipoints (List<LocalPoints>) : a list of local join points aspect (LocalAspect) : the aspect itself

removeAspect(cref, ipoints, aspect)

description	Removes an aspect of a container
parameter	cref (ContainerRef) : the container on which the operation is executed ipoints (List<LocalPoints>) : a list of local join points aspect (LocalAspect) : the aspect itself

addAspect(site, ipoints, aspect)

description	Adds an aspect to a space
parameter	site (URI) : the identifier of the core ipoints (List<IPoints>) : a list of join points aspect (GlobalAspect) : the aspect itself

removeAspect(site, ipoints, aspect)

description	Removes an aspect from the space
parameter	site (URI) : the identifier of the core ipoints (List<IPoints>) : a list of join points aspect (GlobalAspect) : the aspect itself

setAspectContext(aspectContext)

description	Get the user defined context
parameter	aspectContext (Properties) : the properties set for the aspect context

getAspectContext()

description	Get the user defined context
parameter	

Appendix B.

Development Process

B.1. Version Control

When developing a complex application with many different people, it is essential to have some sort of version control over the sources. Version Control enables developers to have a consistent view over the project, by providing mechanisms for internal coordination like the administration of revisions or exclusive access to a file. This gets even more important when the number of developer increases. In case of an open source project, where nearly everyone can produce some piece of code, it is impossible to monitor the project without version and permission control. In addition to that developers have to define their own process, how the version control system is used. E.g., at which status code is allowed to be checked in or when developers are allowed to lock certain files. Besides those rules, there are two fundamental arrangements.

- Never check-in auto-generated code
- Only commit code, which compiles

The most popular systems are the concurrent versions system (cvs)¹ and subversion (svn)². Svn has many advantages over cvs. Subversion duplicates the sources in a hidden folder (.svn), which increases size of the project, but leads to features that are essential to developers. It's possible to show local changes without connection to the repository or to transmit only changed files, which is much more important. In addition to that, committing to a subversion repository is atomic, so that all file changes were saved in the repository or none. Like cvs,

¹<http://www.cvshome.org/>

²<http://subversion.tigris.org/>

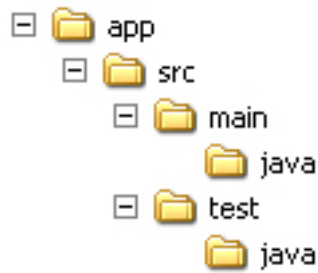


Figure B.1.: Maven Folder Structure

svn is available for most platforms and fits well in many integrated development environments (IDE). Besides the console tools, there are also graphical tools for both systems like TortoiseSVN³ or TortoiseCVS⁴. During the implementation process of MozartSpaces we used svn as version control system, mainly because of the advantages mentioned above. When using eclipse⁵ as IDE there is also a subversion plug-in called subclipse⁶, which enables graphical svn control out of the IDE.

B.2. Build System

In order to have a clean development environment a build system was used during the whole implementation. Ant⁷ and maven⁸ are the most popular ones, when programming in java. Since maven automatically provides a clean separation between code, resources and test packages as shown in figure B.1 the MozartSpaces development team used this as a build system. In addition to that maven provides automatic dependency resolution, which was also important. It also integrates well in common IDE such as eclipse⁹ or IntelliJ¹⁰.

³<http://tortoissvn.tigris.org/>

⁴<http://www.tortoisecvs.org/>

⁵<http://www.eclipse.org/>

⁶<http://subclipse.tigris.org/>

⁷ant.apache.org

⁸maven.apache.org

⁹www.eclipse.org

¹⁰www.jetbrains.com/idea/