Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !INFORMATIK

Design and Implementation of XcoSpaces, the .Net Reference Implementation of XVSM

Core Architecture and Aspects

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Thomas Scheller
Matrikelnummer 0225689

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuerin: Ao.Univ.Prof. Dipl.-Ing. Dr. eva Kühn

Wien, 01.09.2008

____________________ (Unterschrift Verfasser/in)

____________________ (Unterschrift Betreuer/in)

Abstract

With a continuously growing number of computers that are connected over distributed networks, in particular the internet, there is also a growing need for applications to communicate with each other over these networks. Following the idea of shared data spaces, *XVSM* (eXtensible Virtual Shared Memory) provides a middleware solution that allows applications to collaborate with each other in an easy and natural way without the need of a central server. XVSM specializes in the coordination of data, providing flexible data structures for all different kinds of communication, which most other shared data space implementations lack. Despite of that, XVSM aims to be very light-weight, and at the same time provide a great amount of extensibility for easily adding features that are not initially a part of XVSM.

This document focuses on the *XVSM core*, which is the software component that implements XVSM, and introduces its architecture in detail, which aims to provide concurrency, scalability and extensibility for all the features defined in the XVSM model. An implementation of this model is shown with *XcoSpaces*, the .Net reference implementation of XVSM, which has been built following the introduced XVSM core architecture. It is also shown how XcoSpaces has been built as a component oriented architecture that greatly supports the extensibility of the XVSM model from an implementation point of view.

For proving how easy it is to add additional functionality to XVSM, it is shown with XcoSpaces how new features are added that enable simple security mechanisms for the space.

Kurzfassung

Mit einer ständig steigenden Anzahl von Rechnern die über verteilte Netzwerke, vor allem das Internet, miteinander verbunden sind, müssen Anwendungen auch immer öfter über solche Netze hinweg miteinander kommunizieren. Dem Paradigma der „Shared Data Spaces“ folgend, bietet *XVSM* (eXtensible Virtual Shared Memory) eine Middleware Lösung, die es Anwendungen erlaubt, miteinander auf einfache und natürliche Art und Weise zu kollaborieren, ohne die Notwendigkeit eines zentralen Servers. XVSM ist spezialisiert auf die Koordination von Daten, es stellt flexible Datenstrukturen für verschiedenste Arten der Kommunikation bereit, die von den meisten anderen Systemen dieser Art nicht zur Verfügung gestellt werden. Weiters ist XVSM besonders auf Leichtgewichtigkeit ausgerichtet, bietet aber gleichzeitig auch eine starke Erweiterbarkeit, um weitere Funktionen leicht hinzufügen zu können, die ursprünglich nicht in XVSM enthalten sind.

Dieses Dokument konzentriert sich auf den *XVSM core*, die Softwarekomponente die XVSM repräsentiert, und stellt dessen Architektur im Detail vor, die insbesondere darauf abzielt, für die im XVSM Modell definierten Funktionen Erweiterbarkeit, Skalierbarkeit und Parallelität zu garantieren. Eine Implementierung dieses Modells wird vorgestellt anhand von *XcoSpaces*, der .Net Referenz-Implementierung von XVSM, die nach dem präsentierten Modell entwickelt wurde. Weiters wird die komponentenorientierte Architektur vorgestellt mit der XcoSpaces entwickelt wurde, die die Erweiterbarkeit des XVSM Modells aus Implementierungssicht unterstützt.

Um zu beweisen wie einfach zusätzliche Funktionalität zu XVSM hinzugefügt werden kann, wird anhand von XcoSpaces gezeigt, wie der Space um neue Funktionen erweitert wird, die ihn mit einfachen Sicherheitsmechanismen ausstatten.

Acknowledgements

The definition of the XVSM core and implementation XcoSpaces would never have been possible without the help of several people, who should not be left unmentioned.

First I want to thank my supervisor *eva Kühn* for her continuous and hard work on the definition of XVSM, and especially for offering me this great master thesis topic, as well as *Richard Mordinyi* for lots of constructive criticism that helped forming this master thesis.

Special thanks also go to *Ralf Westphal* and *Geri Joskowicz*. Ralf, being an expert for software development and especially the .Net platform, influenced the design of the .Net implementation with very helpful and constructive ideas and built several applications using and testing the .Net implementation. Geri, having great experience in the field of banking applications and with software security concerns in general, helped reviewing the .Net implementation and bringing in new and different ideas concerning XVSM from a business point of view.

I also want to thank the development team of MozartSpaces (the Java implementation of XVSM), consisting of *Christian Schreiber* and *Michael Pröstler*, for their collaboration on defining XVSM and often bringing in different points of view.

My biggest thanks go to *Markus Karolus*, who developed XcoSpaces together with me, and is not only a great programmer never giving up before a satisfying solution is found no matter how insoluble a problem seems, but also a great friend whom I could always rely on.

Content

1 Introduction 9
1.1 An Introduction to XVSM 10
1.1.1 The XVSM Paradigm 10
1.1.2 Other Communication Methods 11
1.2 Objectives and Overview 12
2 A Classification of Space Based Computing Systems 14
2.1 Space Based Computing Systems 14
2.1.1 Blitz (JavaSpaces) 14
2.1.2 GigaSpaces 14
2.1.3 LighTS 15
2.1.4 Corso 15
2.1.5 Others 15
2.2 Classification Structure 16
2.3 Coordination Concepts 17
2.3.1 Coordination Types 17
2.3.2 Meta Data 18
2.3.3 Space substructure 18
2.3.4 Data Types 19
2.4 Operations 20
2.4.1 Basic Operations 20
2.4.2 Transactions 20
2.4.3 Events 21
2.5 Extensibility 22
2.5.1 Coordination 22
2.5.2 Other Concepts 22
2.6 Architecture 23
2.6.1 System Architecture 23
2.6.2 Network Architecture 24
2.6.3 API 24
2.6.4 Security 25
3 The XVSM Core 26
3.1 Introduction 26

3.2 Core Requirements 27
3.2.1 Non-Functional Requirements 27
3.2.2 Functional Requirements 29
3.2.3 Non-Core Functions 31
3.3 Containers 32
3.3.1 Coordination Type 33
3.3.2 Selector 33
3.3.3 Entry 34
3.3.4 Container Properties 34
3.3.5 Container Operations 35
3.3.6 Transactions and Concurrency 37
3.3.7 Meta Container 38
3.4 The Core Architecture 39
3.4.1 Core Containers 40
3.4.2 The Embedded API 42
3.4.3 Remote Communication 43
3.4.4 The Core Processor 45
3.4.5 Event Processing and Timeout Handling 47
3.4.6 Error Handling 50
3.5 Architecture Variants 51
3.5.1 Peer 51
3.5.2 Client 52
3.5.3 Standalone 52
3.6 Profiles 53
4 The XcoSpaces .Net Kernel Implementation 55
4.1 Introduction 55
4.2 The Kernel Architecture 55
4.2.1 The Embedded API: XcoKernel 57
4.2.2 Remote Communication 58
4.2.3 Core Containers 58
4.2.4 Message Types 59
4.2.5 The CoreProcessor 63
4.2.6 Event Processing and Timeout Handling 66
4.2.7 Containers and Coordination Types 67
4.2.8 Transactions 70

4.2.9 Error Handling 73
4.2.10 Logging 73
4.2.11 Documentation 75
4.3 Implementation based Kernel Design 75
4.3.1 An Introduction to Contract First Design 76
4.3.2 Contract First Design in the Kernel 78
4.3.3 The Kernel Component Structure 78
4.3.4 The Microkernel 80
4.3.5 Component Deployment 82
4.4 Contracts 83
4.4.1 Selectors 83
4.4.2 Logging 87
4.4.3 ThreadDispatcher 88
4.4.4 CommunicationService 89
5 Aspects in the XcoSpaces Kernel 92
5.1 Introduction 92
5.2 Kernel Implementation 93
5.2.1 Core Processor 93
5.2.2 XcoKernel 94
5.3 Implementing Aspects for the Kernel 95
5.4 Aspect Implementation Example 98
5.4.1 Introduction – The SimpleSecurityAspect 98
5.4.2 Implementation of the SimpleSecurityAspect 98
5.4.3 Using the SimpleSecurityAspect 100
5.4.4 More Ideas for the SimpleSecurityAspect 101
5.5 Notifications in the XcoSpaces Kernel 102
5.5.1 Introduction 102
5.5.2 The Notification1 Definition 102
5.5.3 The Aspect Based Notification Implementation 103
6 Future Work 106
7 Conclusion 107
8 Appendices 109
8.1 Figure Index 109
8.2 Code Example Index 110
8.3 Abbreviations 110

9 References 111

1 Introduction

In the last years, the number of people connected together by the internet has been continuously growing. In the end of 2007, more than 1.3 billion people were using the internet [1]. Together with that comes also a growing need for people to collaborate with each other online and in real-time.

Concerning software applications, many things have of course changed in the last years as well, from batch applications in the beginning to global and interactive OLTP applications[1]. But still, as Ralf Westphal states in [2], applications are mostly centralized, meaning their data as well as their application logic is stored on a centralized server. With this kind of centralism comes a form of work that hasn't changed as well: Data is processed by people sequentially. Someone inserts data into the system, another one loads and changes it, and so on. Many people may work on the entire amount of data concurrently, but in the end they all do it isolated from each other.

The problem why this form of application is still the most common one is not that developing applications in another way would be impossible; there are enough applications showing that this is not the case. Take Skype [3] for example, which is based on real-time collaboration and is working (at least mostly) peer-to-peer based without a central server. The problem is that the communication paradigms that are used nowadays are all based on the same principles: There is one entity that offers some kind of service (the server), and there are one or more entities that want to use these services (the clients). This is clearly not a good basis for creating applications where entities collaborate with each other over a distributed network, because there simply is no *collaboration* in this form of communication. Furthermore, a distributed application of course must deal with the problem that when many entities need to communicate with each other they either need to maintain many connections between them or a centralized server is needed that needs to maintain connections to all entities (see Figure 1). A much more natural form of communication would be needed, for giving up on centralism and the client-server principle; a form of communication that aims for *serverless real-time online collaboration*:

- *Serverless*: Applications don't need any central infrastructure for communication, but instead they span up ad hoc networks for communicating with each other.
- *Real-time*: Users collaborate with each other at the same time. Only in this way *Collaboration* (in its simplest form e.g. a chat) is possible. Tasks that require fast reaction, where a flow of communication needs to be established, are best done in real-time (in the given context meaning *soft real-time*[2]).
- *Online*: Teams are more and more often distributed around the globe. When such a distributed team needs to come together, this is normally easiest by using a network like the internet, and thereby meeting *online*.
- *Collaboration*: Communication is not based on one entity issuing commands and another one responding, but by the entities collaborating with each other by working on the same distributed data structures (having a *team* instead of a *hierarchy*).

[1] Online transaction processing, or OLTP, refers to a class of systems that facilitate and manage transaction-oriented applications.

[2] Soft real-time systems are typically used where there is some issue of concurrent access and the need to keep a number of connected systems up to date with changing situations. In contrast to *hard real-time*, no hard constraint is defined to the meaning of *real-time*.

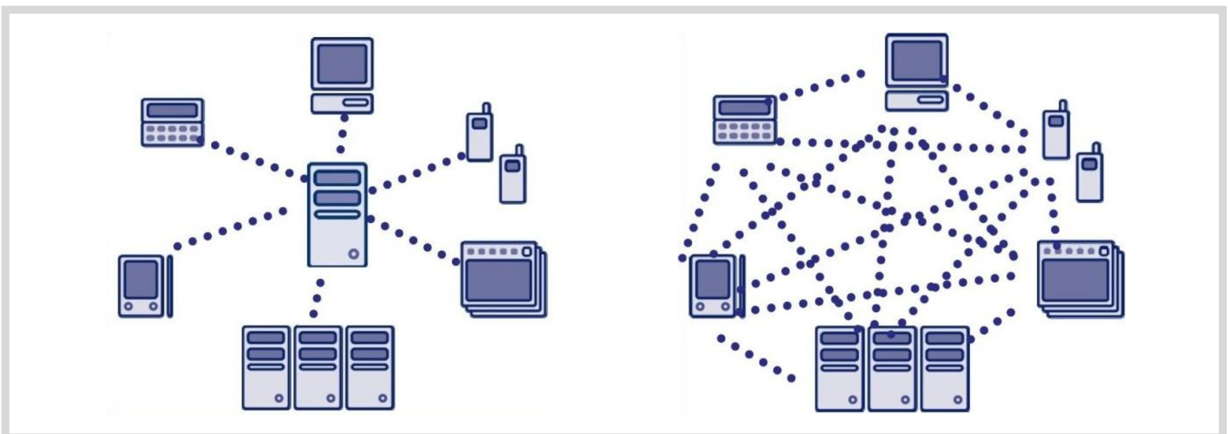


Figure 1: Classical forms of communication in a distributed system: (1) client-server communication and (2) direct communication.

1.1 An Introduction to XVSM

The technological paradigm that aims to implement this form of communication is called *XVSM* (eXtensible Virtual Shared Memory) [4]. It provides a middleware[3] solution based on the principles of *shared data spaces*.

1.1.1 The XVSM Paradigm

A shared data space provides distributed applications with a buffer for passive data, equipped with means for synchronization and coordination of processes. The shared buffer is comparable to a shared variable, a shared memory area or a shared file, and is used to hold various kinds of data.

The idea of a shared data space was first created by David Gelernter in the 1980s [5]. He introduced a coordination language called *Linda* which operates on an abstract computation environment called *tuple space*. Concurrent processes of a distributed application coordinate themselves by communicating with the tuple space. Coordination is performed by writing and reading data tuples to/from the space. The communication always takes place between the processes and the space. This way the sending process does not need to know about the receiving process and there is no need for both processes to be connected at the same time. This decoupling of processes in both time and space takes away a lot of complexity in creating distributed applications. Gelernter calls this communication paradigm which is both decoupled in space and time *generative communication*.

The original Linda model requires four operations that individual workers perform on the tuples and the tuple space:

- **in** atomically reads and removes (consumes) a tuple from the tuple space
- **rd** non-destructively reads a tuple from the tuple space
- **out** produces a tuple, writing it into the tuple space
- **eval** creates new processes to evaluate tuples, writing the result into the tuple space

In addition to that, XVSM adds some functionality that a traditional tuple space lacks:

[3] Middleware is a software layer that lies between the operating system and the applications on each side of a distributed computing system in a network.

- An XVSM space is targeted to run serverless. Instead of running on a server, it is built out of the participating clients themselves and the space's data is divided upon them (see chapter 3.1). (It is of course still possible to let the space run on a single server.)
- The XVSM space supports the use of coordinated data structures like a queue or stack, which allow ordering the elements in the space (introduced in chapter 3.3).
- The XVSM space provides a great amount of extensibility in a way that additional behavior can be injected into the space at runtime (meaning no recompilation is needed), e.g. concerning security or persistency mechanisms (an overview about extensibility in XVSM can be found in chapter 3.6).
- The XVSM space provides an open communication protocol that is programming language independent, which allows that different applications written in different programming languages are able to use the same space and thereby collaborate with each other. The protocol is not described in more detail in this thesis, for more information on it see [6].

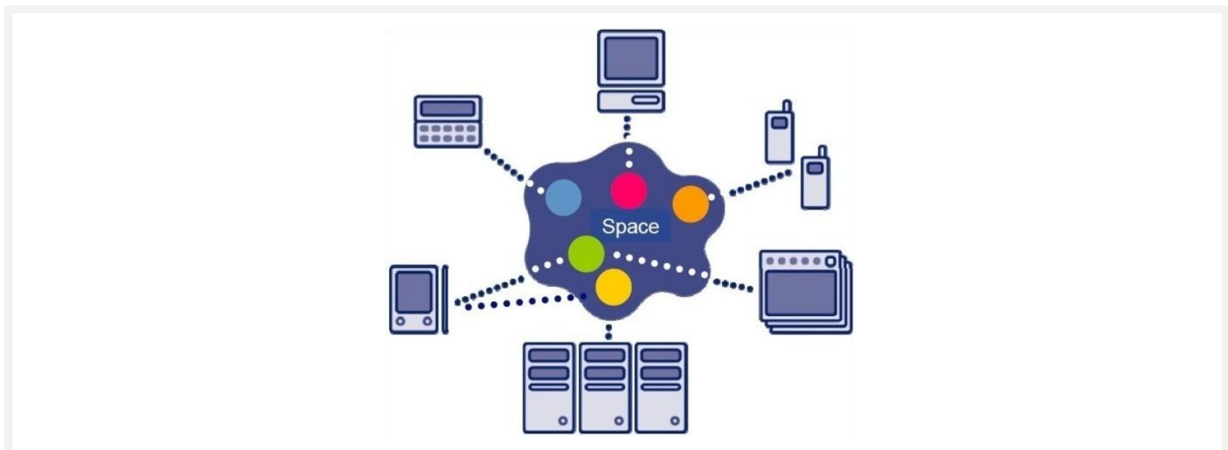


Figure 2: Serverless communication with a space.

1.1.2 Other Communication Methods

This chapter gives a short overview about other common communication methods (as overviewed in [7]).

1.1.2.1 Message Passing

Message passing can be viewed as the ancestor of distributed interactions. Message passing represents a low-level form of distributed communication, in which participants communicate by simply sending and receiving messages. Although complex interaction schemes are still built on top of such primitives, message passing is nowadays rarely used directly for developing distributed applications, since physical addressing and data marshalling become visible to the application layer. Also clear disadvantages over shared spaces are that all communicating entities must know each other (no location transparency) and that they must be available, meaning one entity sending and the other one receiving, at the same time (no time decoupling).

A more advanced approach of message passing is using message queues. With this approach, the producing entity sends the message not directly, but places it into a queue, from where it is later taken out by the consuming entity. Since the entities don't communicate directly with each other, they are decoupled in both space (don't need to know each other) and time (message doesn't need

to be consumed immediately). Popular messaging services are for example JMS (Java Message Service) [8] and MSMQ (Microsoft Message Queuing) [9].

1.1.2.2 RPC

One of the most widely used forms of distributed interaction is the remote invocation, or *remote procedure call* (RPC), an extension of the notion of "operation invocation" to a distributed context. A remote procedure call is initiated by the client sending a request message to a known remote server in order to execute a specified procedure using supplied parameters. A response is returned to the client where the application continues along with its process. While the server is processing the call, the client is blocked.

By making remote interactions appear the same way as local interactions, the RPC model and its derivatives make distributed programming very easy. This explains their tremendous popularity in distributed computing. Distribution cannot, however, be made completely transparent to the application, because it gives rise to further types of potential failures (e.g., communication failures) that have to be dealt with explicitly.

Additional problems with RPC are that there is no decoupling in space and time (since the local interaction model that RPC follows doesn't need these things) because client and server are communicating directly with each other, and that (in contrast to shared spaces) communicating entities are always pressed into the client/server role, with the server processing calls from the client and answering to them.

Popular RPC implementations are for example Java RMI [10] and .Net Remoting [11].

1.2 Objectives and Overview

The main topics of this thesis are the detailed definition of the XVSM model and all of its features, and the implementation of XVSM on the *.Net* platform. Because this is a task that is far too large to be handled by one master thesis alone, the topics have been split up into several pieces among four participating master thesis students: *Markus Karolus* and *Thomas Scheller* (me) were responsible for developing the XVSM .Net implementation called *XcoSpaces*, *Christian Schreiber* and *Michael Pröstler* had the additional task of developing an implementation of XVSM in Java (called *MozartSpaces*), while all four took part in designing the theoretical XVSM model equally.

The theoretical topics of the XVSM model have been split up between the master theses of the four students ([12], [6], [13] and this one), while the topics concerning one certain implementation have been split up between the two students responsible for that implementation (in my case the .Net implementation). Therefore, only all for master theses combined give a complete overview over the XVSM model and both the Java and the .Net implementation. However, every thesis focuses on a set of topics that lets it come self-comprehensive.

This thesis first takes a look at systems related to XVSM that are based on the shared data spaces paradigm (see chapter 2), and introduces a classification structure which is then used to compare these systems to each other and to show where they are equal to or different from XVSM. After that, the document introduces the architecture of the *XVSM core* (which is the piece of software that represents XVSM). Chapter 3 first focuses on the requirements and functionalities of the XVSM core and then introduces the core architecture that is built to fulfill all of these requirements. Chapter 4

then focuses on how this architecture has been realized in the XVSM .Net implementation of the core called *XcoSpaces*, and also describes the modular and extensible structure of the software itself in detail. Finally, chapter 5 deals with the extensibility features of XcoSpaces in detail and gives an example of how to use these features in practice.

A point that should also be mentioned is that XVSM, as well its implementation XcoSpaces, is under constant development. Although both are in a "completed" state as presented in this document, they are continuously revised and improved. This document describes version 0.9.1 of XcoSpaces, and the XVSM model as of January 2008.

2 A Classification of Space Based Computing Systems

There is already a whole bunch of middleware solutions available that are based on the idea of shared data spaces like XVSM is. But although they may have that in common, most approaches are clearly different and have other target areas of use. The goal of this chapter is to show which features can be seen as most important in systems that are using the idea of shared spaces, and survey some of the most important models and implementations concerning which of these features they implement. Part of this analysis is to create a general structure that can also easily be used later to classify other space systems and models.

Chapter 2.1 gives an introduction of the most common space implementations. While chapter 2.2 introduces a structure for classifying space systems, chapters 2.3 to 2.6 describe the aspects of this classification structure more detailed and use it to compare the previously introduced space systems to each other.

2.1 Space Based Computing Systems

There are a lot of implementations and models available that make use of the idea of shared data spaces and would be interesting to research. This chapter tries to give an overview of the most common of these systems, whose features will then be compared and classified in the following chapters.

2.1.1 Blitz (JavaSpaces)

Blitz [14] is an open source implementation of JavaSpaces [15], which is a specification of a Linda-like space in Java (Blitz can thereby be seen as a representation of JavaSpaces in this paper). In addition to the typical JavaSpaces functions (which include writing entries to the space and reading or taking them using tuple matching, all of that with transaction support and the possibility to define lease times for single entries, as well as getting events from the space) Blitz also supports persistency mechanisms using an embedded database.

Blitz is also highly configurable and provides a set of supporting tools that allow monitoring and administrating the space and generating statistics.

2.1.2 GigaSpaces

GigaSpaces [16] is also an implementation of JavaSpaces, but actually provides its own API on a higher level that supports more functionality and a better usability than JavaSpaces do. In addition to the basic functionality of JavaSpaces (see Blitz), the most important features are the following:

- GigaSpaces contains the *OpenSpaces* framework which is a Spring-based framework [17] built on top of GigaSpaces. This allows components (called *processing units*) to easily scale out across multiple machines, using the space for both managing data and communication between processing units.

- The GigaSpaces server can be clustered and supports all common mechanisms for caching, replication and persistence, and provides high performance and scalability that is very well suited for large enterprise applications.
- Provides extensive tool support for administration, monitoring and statistics.
- Provides a detailed security model for authentication, authorization and data encryption.

2.1.3 LighTS

LighTS [18] is a Linda tuple space implementation written in Java. It has been designed with the following goals in mind:

- *Minimal support*: LighTS implements just Linda operations. No persistency, security, or remote access are provided. All of this can be built around the core provided by LighTS.
- *Lightweight, fast processing*: LighTS needs no runtime support (the tuple space is just an object that gets shared among Java threads in the same JVM), and through its minimality performs well with small numbers of tuples.
- *Extensibility*: LighTS is designed to be as extensible as possible.
- *Small footprint*: The packages providing the LighTS tuple space fit in 11 kb of jar file, which is a desirable property when the target platform includes hand-held and palmtop computers.

2.1.4 Corso

Corso [19][20] can be seen as the predecessor of XVSM, as the idea for this system has also been developed at the TU Vienna institute for computer languages. In contrast to the previously introduced systems, Corso isn't based on the idea of Linda tuple spaces, but more on the principle of virtual shared memory. It allows the sharing of data objects that allow writing data into them / reading it from them. Transactions are supported for accessing these data objects and making them persistent (using an embedded database), as well as different replication mechanisms for distributing data objects over a network of Corso kernels. Corso is written in C and supports APIs for access over C++, Java and .Net.

2.1.5 Others

Although their features are not investigated further in this document, some other systems that are based on shared data spaces should also be mentioned:

- *Outrigger*, which is part of *Jini* [21], is the standard implementation of JavaSpaces from Sun and therefore very similar to Blitz.
- *TSpaces* [22], developed by IBM, is like JavaSpaces a Linda-like space implementation in Java, and is also very similar to JavaSpaces in its functionality and behavior.
- *Lime* [23] ("Linda in a mobile environment") also implements the Linda tuple space model, but especially focuses on the mobility of hosts (entities hosting a space) and agents (entities accessing spaces) and extends tuple spaces with a notion of location.
- *TuCSoN* [24] ("tuple centres over the network") exploits a notion of local tuple-based interaction space, called *tuple centre*, which is a tuple space enhanced with the notion of behavior specification. By programming its behavior in response to communication events, a tuple centre can embody coordination laws.
- *Mars* [25] ("mobile agent reactive spaces") as well defines Linda-like tuple spaces that can be programmed to react with specific actions to the accesses made by mobile agents.

2.2 Classification Structure

To classify different space systems and compare them to each other it is first necessary to actually know for which concrete aspects such a system must be surveyed. The methodology loosely follows the ideas for surveys and classifications as presented in Androutsellis-Theotokis' and Spinellis' survey of peer-to-peer content distribution technologies [26] and Eugster's classification of publish/subscribe systems and other communication methods [7]. The classification concentrates mostly on criteria that also allow classifying systems that have not yet been implemented, which is often the case when it comes to research work. The result of the research of many models and implementations of shared data spaces is the classification structure as shown in Figure 3:

Probably one of the most important properties of a space is how it coordinates data, in other words its *coordination concepts*. This includes what possibilities there are to store data in the space, and what possibilities to get it out again. It is also very interesting if the data in the space can be structured in any way (e.g. hierarchical), if there are any possibilities store meta data, and what types of data can actually be stored in the space.

Another important property of a space is the *operations* it supports. Basic operations are writing and reading data, but in many situations these are not satisfying, so additional operations are provided. Differences are also in the abilities of the operations themselves, e.g. if they support transactions or not, or if it is possible to execute bulk operations. Another aspect here is the possibility of getting events from the space since it can be very important to recognize when data in the space has changed.

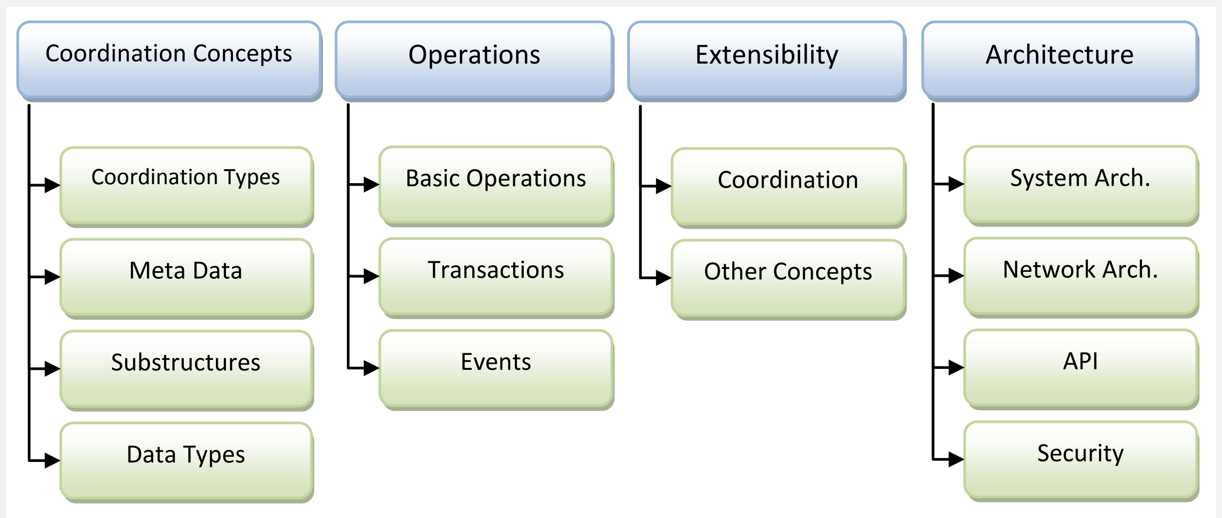


Figure 3: Classification Structure for Space Based Computing Systems.

A property of a space that is rather difficult to classify but is nevertheless very important is its *extensibility*. Extensibility can concern nearly all aspects of the space, probably one of the most important being its coordination concepts. It is also interesting how extensibility is guaranteed and how difficult it is to extend the space by new features.

The *architecture* of a space also has some properties of interest: One is how the space is deployed, which could be either embedded within an application or completely standalone (the system architecture), or how the space manages itself over the network, which could be either client/server based or peer-to-peer based (the network architecture). The space's APIs are another part of the

architecture, seeing it from the point if the space supports standardized APIs and if it has an open communication protocol that can be used for accessing the space. Security is also a very important point here (it is in general a very important thing in distributed systems), concerning user authentication, authorization and encrypted communication.

The following chapters describe these properties of a space in more detail and compare the previously introduced space systems to each other.

2.3 Coordination Concepts

2.3.1 Coordination Types

A space's coordination types deal with the question of how data is coordinated within the space, and by that how data is written to and read from the space. A coordination type that nearly all space systems support is *Linda* (which is not surprising since most spaces originate from the Linda tuple space model), in which case every object written into the space is a tuple. When reading, a "query tuple" has to be defined, that is used as a template for matching against the tuples in the space. With this coordination type, the tuples in the space are completely unordered and thereby independent from each other, which makes it comparatively easy for the space to handle concurrent operations.

Although Linda is suitable for many usage situations, with the data being completely unordered there are some issues that can hardly be realized with it, above all coordination types that use an *implicit order*. Such a coordination type is *fifo* (first in first out) which automatically orders data like in a queue (the first element that has been written is also the first one to be read) and is therefore especially useful in producer/consumer-like scenarios where data should be processed in the order it is written into the space. Another example of an implicit order would be *lifo* (last in first out), where elements are ordered like in a stack (the last element that has been written is the first one to be read).

The third kind of coordination type is one that defines an *explicit order*, meaning the order can be influenced when writing an element into the space, and/or a certain element can be picked out using the given ordering when reading elements from the space. This type of coordination especially includes some data structures that are very often used in programming, namely arrays, lists and hashtables. In current systems these coordination types are very rare, which is no wonder since from the coordination types mentioned until now they are the least combinable with Linda, which most systems use as their base coordination type.

Blitz, GigaSpaces and LighTS, as they implement either JavaSpaces or the Linda tuple space model directly, have Linda as their main coordination type. Blitz and GigaSpaces also support a fifo ordering for the entries in the space. In GigaSpaces, this is simply done by marking a class with an *@Fifo* annotation which automatically leads to ordering every object of this class that is written into the space in fifo order. In addition to that, GigaSpaces (surely being the most matured of these systems) supports defining a key, which means the space can be used like a hashtable. By marking a field in a class as key, objects of this class within the space get an identity, which means no two objects with the same key can be written in to the space, and an object in the space can be updated accessing it with its key.

Corso goes a way that is quite different to the other systems. The shared objects that can be stored in Corso have a unique identifier and can be given a unique name. The unique identifier allows accessing the shared object, and can only be acquired when knowing the shared object's name (or when you are the one that created the shared object). So coordination is kind of similar to a hashtable, with the shared objects having a name as key (or no key at all). Despite of that, there are no other coordination types in Corso.

XVSM is a system that especially concentrates on the coordination of data in the space. Therefore an XVSM space is structured into *containers* which allow very flexible coordination, including coordination types with both implicit and explicit order. Containers are introduced in chapter 3.3.

2.3.2 Meta Data

Meta data of a space can be seen as information about the data in the space (e.g. the number of entries in the space) or information about the space itself (e.g. the name of the space, if it has one). This is a feature only rarely supported by the given space systems. GigaSpaces and LighTS support getting the number of entries in the space or the number of entries matching a certain template. LighTS also supports assigning a certain name to the space, which can help distinguishing spaces from each other when using more than one space (which is easy with LighTS since the space is just a Java object so there can easily be multiple instances, but rather unnecessary for other systems, where there is normally only a single instance of a space running on one machine).

This is again a point where XVSM tries to stand out above the other systems by providing a well defined meta data structure. In XVSM, every container is assigned a so-called *meta container*, that stores meta data for this container. An introduction of meta containers can be found in chapter 3.3.7.

2.3.3 Space substructure

In addition to storing elements, some spaces allow data to be split into groups of elements belonging together, or for elements to be subordinated to others. This is what is meant by the word "substructure", structures that are below the scope of the space, but still above the scope of a single data element. A simple example would be a space that can be split into sub-spaces (smaller spaces within the space itself), which can store entries independent from other sub-spaces, and thereby also allow handling the entries in different spaces completely independent from each other.

Since a typical property of a Linda tuple space system is that the entries in a space are completely unordered, the other systems don't give much thought about substructures in the space such as XVSM and Corso have. It is not possible to store tuples (or *entries*, as they are called in JavaSpaces) in a hierarchical order. However, it should be mentioned that the JavaSpaces implementations (Blitz and GigaSpaces) support inheritance, meaning: If class B is a subclass of class A, and there are both instances of A and B stored in the space, reading data from the space with a template of class A will not only return matching instances of A, but also of B. When having a well defined inheritance structure, this allows querying for all data in the space (using the base class), or restrict queries only to certain data (using a sub class), which grants a structuring of the space to a certain degree.

XVSM supports this feature by the use of its *containers* (see chapter 3.3) which automatically give a structure to the space since no data can be stored in the space directly but only in its containers, and can also be used hierarchically (a container containing references to other containers).

2.3.4 Data Types

It is of course not only important how data is stored in the space, but also what kind of data can be stored, if there are certain restrictions or if the data must be in a predefined form. In this point the presented systems have rather different approaches.

Typically for Linda, data stored in a space needs to be some kind of tuple. LighTS is no exception to that, requiring for all objects that are written to the space to be tuples. There are predefined classes for tuples in LighTS that implement a tuple matching for equality. But LighTS also allows any objects to be written into the space whose classes implement the *ITuple* interface, which makes it possible to write nearly everything into the space (of course with the expense of needing to implement the ITuple interface yourself, which means defining how your class is matched against other ITuples).

In JavaSpaces implementations like Blitz, data written into the space must be an entry (implementing an *Entry* interface). Entries are similar to tuples, with the difference that they look more like normal classes in object oriented programming languages (while a tuple is only a list of values). Blitz simply stores all public fields of an entry in the space, with the restriction that fields must not be primitive types (only object references, which is needed since Blitz uses the *.equals()* method for comparison of the single values).

GigaSpaces provides a much better usability here, allowing very flexible entry definitions by the use of annotations. Classes don't need to implement any interfaces, but can just define if all their public fields or even all their private fields should be stored in the space, and also if certain fields should be ignored. Fields can even be annotated with special information that marks them e.g. as *keys* or *indices*, and according to that these fields will be specially treated within the space. This is also very similar to the mapping definition by annotations in *Hibernate* [27] entities.

GigaSpaces even supports interoperability for data written into the space, meaning for example data written into the space by a .Net program can be read by a Java program. The only requirement is that both sides have a class that corresponds to the same data structure within the space. GigaSpaces therefore provides additional annotations that allow e.g. adding to the .Net class information about the complete name of the Java class and adding information to the properties about the names of the according fields in the Java class so that the space can internally recognize both classes as the same.

XVSM is normally not intended to be used with tuple matching, and therefore doesn't require all data in the space to be tuples, but allows writing data of any type into the space (it only has to be serializable). Restrictions are only given when either using tuple matching (in which case tuples must be used), or when interoperability is needed (currently interoperability is granted between the Java and .Net version of XVSM), in which case only primitive types and tuples are supported.

Last to mention here is Corso, which only allows writing primitive data types and strings (a sequence of these types with unrestricted length can be written into a single shared object). With this restriction, unfortunately binary data also has to be converted into a string, which is possible but not very efficient as it consumes much more memory than byte data. A special thing about Corso is that a shared object can either be *const* or *var*. The content of a *const* (= constant) can only be written once and is from then on not changeable any more, in a *var* (= variable) the content can be written any number of times.

2.4 Operations

2.4.1 Basic Operations

The most basic operations that spaces support are of course writing data into the space and reading/removing it from the space. But not only these operations are often provided in different flavors, most systems also provide additional operations to especially support individual functionality. All of the presented spaces support the operations *write*, *read* and *take* (which is a read operation that at the same time removes the read data from the space), except Corso which has no *take* operation. These terms are used in Blitz (as they are defined that way in JavaSpaces), GigaSpaces and XVSM (with the difference that operations in XVSM always target a single container and not the whole space). LighTS, being a traditional tuple space implementation, uses the terms *in*, *out* and *rd* (equal to take, write and read). All of the mentioned systems support a blocking mechanism, meaning when a read or take operation is executed and it cannot be fulfilled directly (e.g. when there are no entries in the space that are matching the given template), the operation blocks until it can be fulfilled. All systems except LighTS also allow defining a timeout value for the maximum time an operation should block.

There are two variants of these operations that are additionally important: First are non-blocking versions of the above operations (supported by all mentioned systems), meaning they don't block when there are no matching entries in the space, and don't throw an error in this case but just return nothing. Second are bulk versions of the above operations (supported by all except Blitz), that allow writing/reading/taking multiple elements to/from the space at once (when reading/taking: all elements that match the given template), which normally has the effect of performing better compared to having to call a method multiple times. GigaSpaces and XVSM additionally support defining an exact count of elements to be read or taken.

GigaSpaces and XVSM support some additional operations: First is an operation that directly removes entries from the space, without returning them (in GigaSpaces called *clear*, in XVSM called *destroy*). Second is an operation to update entries in the space when used with *key* coordination (which both of the systems support). For an introduction of the operations supported by XVSM see chapter 3.3.5.

Again, Corso is very different here. The basic operations are creating and destroying named and unnamed shared objects (it must be decided at creation time if the object is variable or constant), and writing data to or reading it from a shared object.

2.4.2 Transactions

Transactions allow for a group of operations that are performed on a space to guarantee that they are either successful all together or have no effect on the space at all. Typically, transactions guarantee the *ACID* properties [28], and allow the operations *commit* (all changes to the space made by this transaction are tried to be persisted) and *rollback* (no changes made by the transaction are persisted, the state from before the transaction is restored).

Interesting differences in transactions in different space systems can be if they are either *optimistic* or *pessimistic*. A pessimistic transaction keeps locks to all resources that have been accessed by operations within this transaction, assuring that no other transactions can access these resources until the transaction is either committed or rolled back. This may restrict actions for other

transactions, but guarantees that a transaction can be committed because no other ones could interfere with it. In contrast, an optimistic transaction doesn't keep any locks but just checks when committing if all operations performed under this transaction are possible, which would e.g. not be the case when in the meantime another transaction has taken an entry from the space that would need to be taken by an operation of this transaction.

All of the presented systems except LighTS have support for transactions. Blitz, GigaSpaces and XVSM implement a pessimistic transaction model, locking entries that are accessed by a transaction until this transaction is committed or rolled back. This also has an impact on the blocking operations introduced before: If data to be read or taken is currently locked, an operation will block until the lock is released. All of these systems support the multiple reader / single writer model for locking, meaning that entries can be read by multiple readers at the same time, but only be taken by one at a time.

Corso in contrast implements an optimistic transaction model. Every shared object has a version number that automatically increments whenever its content is changed. Data cannot be written to a shared object if something was written to the object since it has last been read (which is recognized with the object's version number having increased). GigaSpaces additionally supports a similar concept together with the *update* operation, by being able to annotate an entry field as *version* field which is automatically incremented when an entry in the space is updated.

Also to mention is that the transaction models of Blitz, GigaSpaces and Corso support *distributed transactions* (transactions that are spread over the network). As defined by JavaSpaces, Blitz and GigaSpaces have a pluggable transaction manager and thereby easily allow to be connected to transactions that also include targets that are completely independent of the space.

For a detailed description of transactions in XVSM see [6].

2.4.3 Events

The possibility to get events (or *notifications*) from the space can be very helpful, because this makes it possible to be informed when there are changes made to the space, e.g. new data is written. All systems except LighTS have support for notifications.

In Blitz and GigaSpaces, the JavaSpaces operation *notify* can be used to register a notification at the space. It allows defining a template, so that whenever an entry matching this template is written to the space the notification is triggered. Callback is done by Java event handlers. As an additional way of receiving notifications, GigaSpaces allows registering a class for a certain event in the space, using a special annotation for the method that should be called when an event occurs. Through annotations it is also possible to define which events should trigger the callback (write or update) and define a certain template.

In Corso, a notification can be registered at a shared object, triggering whenever data is written into the shared object. It is also possible to include more than one object within a single notification. A drawback of notifications in Corso is that they don't use event handlers, but require active waiting, so event handling would need to be implemented as a layer on top of Corso's notifications if needed.

In XVSM, notifications can by default be registered at a certain container, and it can be decided which operations should trigger the notification (read, take, destroy, write and/or shift), but very well

extensible to come in very different flavors. A detailed introduction of notifications in XVSM can be found in [13].

2.5 Extensibility

Extensibility is something that is very difficult to classify, since the concepts of each system for providing extensibility are completely different and thereby difficult to compare to each other. Additionally, these concepts often require a deep knowledge of the systems to really understand what can be achieved with them. So, this chapter tries to give an overview of the extensibility of the presented systems, but does in no way claim to be complete, since it would take an own paper to research only these aspects of the systems in detail.

2.5.1 Coordination

The extensibility of a space's coordination concepts is a very interesting thing, because it could make the space suitable for or adaptable to many more usage situations than with the concepts initially provided. In general extensibility for coordination means being able to change the space's behavior of interpreting, managing and reacting to data.

For implementing different coordination concepts, LighTS allows to implement the tuple class by yourself (the space only uses an *ITuple* interface and not a certain class for tuples). Since the tuple class also includes the matching logic, this also allows implementing the matching logic yourself. In this way any objects that implement the given *ITuple* interface could be written into the space and extend the default Linda tuple matching behavior or even create a completely different matching behavior.

The topic of extensible coordination is something that XVSM is strongly focused on. As already said, data in XVSM is stored in containers, which can be defined for different types of coordination. These types of coordination are extensible in a way that users can implement their own ones and thereby create nearly every behavior thinkable of. Coordination types in XVSM and their extensibility are explained in detail in [12].

The other systems don't provide mentionable extensibility in coordination.

2.5.2 Other Concepts

There are of course many possible ways for providing extensibility for a space. Also, all systems have different viewpoints of what aspects of the space are important to be extensible (which of course depends on the overall goals of the certain system) and how this is best done. A well known concept for providing extensibility are for example *aspects* (coming from *aspect oriented programming* [29]), or *interceptors* (different term, but very similar to aspects), which allow adding pieces of code at certain points in a software, that, after being added, will be executed every time when the execution of the software reaches this certain point, and thereby allowing to add additional behavior or change the existing one.

XVSM makes use of such aspects, and defines a list of points where they can be inserted. More information about aspects in XVSM and its implementation XcoSpaces can be found in chapter 5.

LighTS aims to provide extensibility through its clearly defined space and tuple interfaces (the tuple interface has already been explained above). By the space being accessed over an interface, the

implementation underneath (LighTS calls it the *tuple space engine*) can easily be replaced by an own one without having any influence on the application using it. Though this provides many possibilities, completely replacing the logic of LighTS and only using its interfaces doesn't seem to have that many advantages despite of sticking to a standardized API. The LighTS interfaces can also be used to build adapters on top of other space systems, so that by using the LighTS interfaces, the space underneath can simply be replaced by another one.

GigaSpaces doesn't provide possibilities for influencing the behavior of the space. What is most mentionable about GigaSpaces concerning extensibility is its functionality for hosting components. A GigaSpace can host components (small applications), called *processing units*, and manage them in a Spring based framework. By that, the components are completely independent from each other, and an application hosted by the space can easily be extended by new components without having to change existing ones, or when the load increases more of the already existing components to handle the additional load can also be added without difficulty. In combination with easily being able to add new GigaSpaces servers to an existing cluster of servers (see network architecture), GigaSpaces scales very well. So it can be said that GigaSpaces' extensibility features mainly concern scalability.

Blitz and Corso don't really provide any features for extensibility. As it is not uncommon for open source projects, Blitz developers state that Blitz is extensible through its well designed class structure, by changing source code yourself.

2.6 Architecture

2.6.1 System Architecture

System architecture deals with questions about how the space is running on a single machine. Main differences here are if a space system is running *embedded*, meaning within the application that uses the space (which has the advantages that the space can be started by the application directly, and that the space can also be accessed directly without any inter-process or inter-network communication), or if the space is running *standalone*, meaning it has to be started and is running completely independent from any applications using it (which is also its advantage). For spaces that can run embedded, the standalone option normally comes for free since this can more or less be done with an application that just starts the space. Showing from the presented systems, embedded solutions are more lightweight and easier to use while standalone ones provide better manageability and administrative tools support.

XVSM and LighTS are embedded solutions, both are by default started directly from the application using them. Both systems are extremely lightweight, with LighTS being the smaller one, but XVSM being superior in functionality. Since both systems are very lightweight, they are also optimal for execution on a mobile environment. Both LighTS and XVSM have support for mobile devices, with LighTS running in *Java Micro Edition* and XVSM in *.Net Compact Framework* (though the mobile solution is still in development and therefore not handled further in this document). While it doesn't make sense for LighTS to be started standalone as it only can be accessed embedded, XVSM additionally supports this option (this depends on the implementation though, XcoSpaces and MozartSpaces do support it). A detailed description of the XVSM system architecture can be found in chapter 3.4.

Blitz, GigaSpaces and Corso are systems that are by default started and running completely standalone. All of them provide tools for administration and management so that the space can be managed completely independent of any applications using it. Blitz and GigaSpaces also have the possibility of being started embedded, but Blitz has some restrictions in functionality concerning transactions when being used that way (GigaSpaces has no restrictions).

2.6.2 Network Architecture

In contrast to system architecture, the network architecture is about how a space is structured within a network. Main differences are whether the structure is *centralized* (typically a client/server structure) or *decentralized* (like in a peer-to-peer structure). Questions that are linked to this are if the space itself is distributed (not necessarily only in a peer-to-peer structure, it could also be that the server is distributed across several machines), and if issues like *caching*, *replication* and *partitioning* are supported.

Typically for the JavaSpaces model, the architectures of Blitz and GigaSpaces are client/server based. Blitz allows having several spaces coexisting in a network, and supports using them concurrently by being able to use distributed transactions and including operations on different spaces within the same transaction (which is of course also possible in GigaSpaces), but other than that implements no mentionable mechanisms for replication or caching. GigaSpaces on the other hand is clearly standing out here: The GigaSpaces server can be clustered and provides replication, partitioning and caching, supporting the most common topologies. GigaSpaces also provides failover mechanisms, so that data from failed instances that are part of the cluster is automatically relocated. Additional servers can also easily be added to the cluster, providing a very well scalable architecture.

Corso (having as well a client/server based architecture) also allows connecting several servers (called *kernels*) together, and provides possibilities for caching and replication of shared objects. For every shared object a replication/caching strategy can be chosen at the time of creation.

In contrast to these systems, XVSM has a pure peer-to-peer structure. An introduction to the network architecture of XVSM can be found in chapter 3.1.

Since LighTS doesn't have any support for remote communication, it has no network structure to review.

2.6.3 API

This point deals with the question if the space provides APIs that follow well known standards and are thereby easily adoptable. Support for standard APIs can always be a great benefit because a user is allowed to use an API that s/he perhaps already has experience with, and can also easily replace the software lying underneath the API if needed. Another interesting question is if the space supports any (programming language independent) open protocols for communication, which would have the benefit of being able to completely replace the space software component with self written software (which must only stick to the protocol for communication with the space).

Both Blitz and GigaSpaces support the JavaSpaces API which is a well known standard (though by using the JavaSpaces API it is not possible to use all of GigaSpaces' functionality). Despite of that, Blitz doesn't have any other standard API support, and is only useable in Java. GigaSpaces in addition also

provides possibilities to use *JMS* (Java Messaging Services) or *RPC* (Remote Procedure Calls) over the space (having wrapper classes that actually use the space underneath for communication).

Like GigaSpaces, the Java Version of XVSM supports a *JMS* API which uses the space underneath for communication. Despite of that, XVSM has an open XML communication protocol. This protocol can be used for remotely communicating with a space and is not platform specific (the protocol is also used for communication between the .Net and Java version of XVSM). Implementing this protocol, any software would be able to talk to an XVSM space. A detailed description of the XVSM xml protocol is given in [6].

LighTS and Corso both don't implement any mentionable standard APIs (despite that the API of LighTS sticks to the Linda standard).

2.6.4 Security

For middleware like space based systems, security is a very important thing. The space could hold sensible data which needs to be protected from being read and altered by users without permission (which is especially important if the space is exposed to the internet and thereby accessible to everyone). The three most important aspects concerning security are *authentication* (user name/password authentication), *authorization* (assigning permissions to certain users to perform specific operations with the space) and *encryption* (of messages that are sent over the network).

Being typically used in large enterprise applications, GigaSpaces has a very mature security model that supports all of these security aspects. GigaSpaces provides a *role* system for user authentication and authorization. *System roles* provide security on an operational basis, meaning it can be defined which operations a role is allowed to execute. *Custom roles* allow a more detailed definition, going down to class/object content level. For example, a role could be defined that blocks read/write of any entry instance from a specific class, or a field value security classification rule where a role blocks a read operation based on a template-specific field value (e.g., permit a user to read Entries from *ClassA* type where their field *A=1*). Classes inherit their security properties from their super classes. Message encryption is granted as well with the use of *SSL*, so the GigaSpaces security model will very rarely leave any desires open.

Corso also provides a built in security solution, though with less functionality than the one of GigaSpaces. Authentication is based on the user accounts of the underlying operating system. The user that started the Corso server automatically is an *administrator* that can do anything within the system. When connecting to Corso, a username and password has to be used. The list of trusted and allowed users can be defined in a configuration file (trusted users may also administer the space, while allowed users may only access its data).

Blitz, LighTS and XVSM have no built-in support for security mechanisms. While Blitz and LighTS also don't make any special considerations of how to add security to their systems, XVSM aims to provide any functionality necessary for adding security by use of its extensibility features. An example for a simple security model built for XcoSpaces can be found in chapter 5.

3 The XVSM Core

This chapter gives a detailed description of the *XVSM core*, which is the software that represents the XVSM space with its core functionality. It first introduces in general what the tasks and requirements of the core are. Then the core's architecture is described in detail, as well as the possible architecture variants and how additional functionality can be added.

3.1 Introduction

A space in XVSM can be seen as a virtual room. It can hold coordinated data structures that are called *containers*. Data can be added to and removed from these containers. And of course new containers can be added to the space and existing ones can be deleted.

The *XVSM core* (further just called core) is the piece of software that creates and manages the space and the containers (and their data) that are part of this space, in other words a space is *hosted* by a core. The core is also responsible for providing possibilities to access the space. In general, it can be said that the core is software that represents the space and therefore has to implement all of the space's functionality (like for example operations on containers, as described in chapter 3.3).

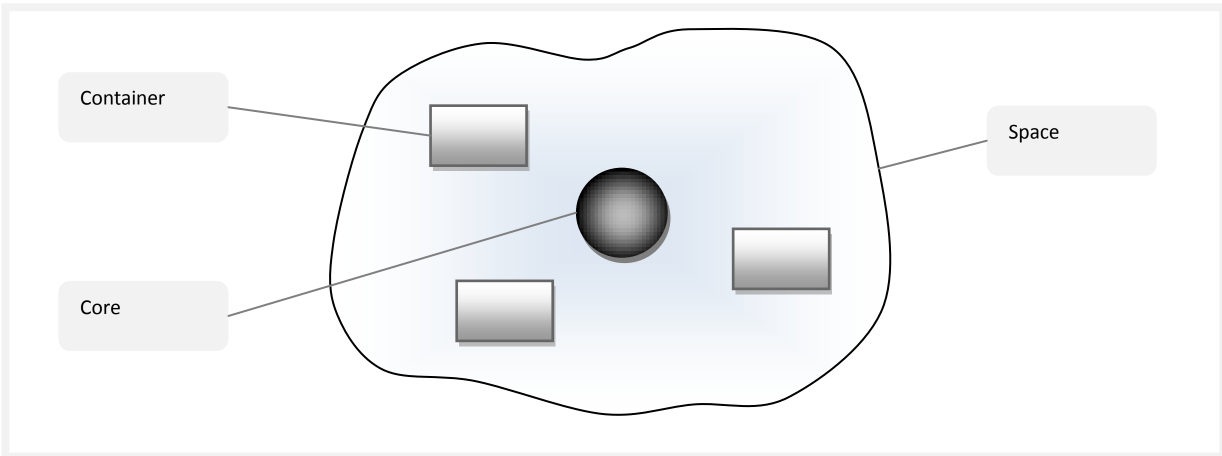


Figure 4: A core that hosts a space with three containers.

Figure 4 shows how a core hosting a space. But this is of course only the simplest scenario, as this point of view is only from a single entity (like a standalone application) that is running a core. But the strength of XVSM comes with collaboration, which means cores communicating with each other and thereby forming a space that goes beyond only the locally stored containers.

XVSM has a pure peer-to-peer structure. A client in XVSM, at the same time also being a server (because it is actually a space itself), is therefore called *peer* (with a single peer of course being an instance of the core). Since being able to just access a space but not participate in it directly could also be useful in different situations (e.g. from a mobile device, not having much resources), a *client only* version of the peer is also supported, which doesn't have a local space itself and thereby cannot store any data (see chapter 3.5).

For better understanding of how the cores are acting in a network, we distinguish between *local* and *remote* cores. The local core is the one that is running locally (managed and used by our own application). A remote core is one that can be accessed from the local core by some kind of remote communication (is doesn't matter which protocol or binding is used, as long as both cores use the same and are thereby able to communicate with each other). In accordance to this definition, the local space is the one that is hosted by the local core (the local space consists of all containers that are hosted by the local core). A remote space is a space that is hosted by a remote core, and consists of all the containers hosted by that core.

By knowing the address of a remote core, the local core can access any container in that core's space. In other words, the local core can not only be used to access containers in the local space, but also to access ones in a remote space as long as its address is known. For the application using the local core, working with a container that is part of a remote space is not any different than working with a container in the local space.

Taking that into account, the *space* from a single core's point of view is the entirety of all containers hosted locally plus all containers accessible by the core that are hosted by a remote space.

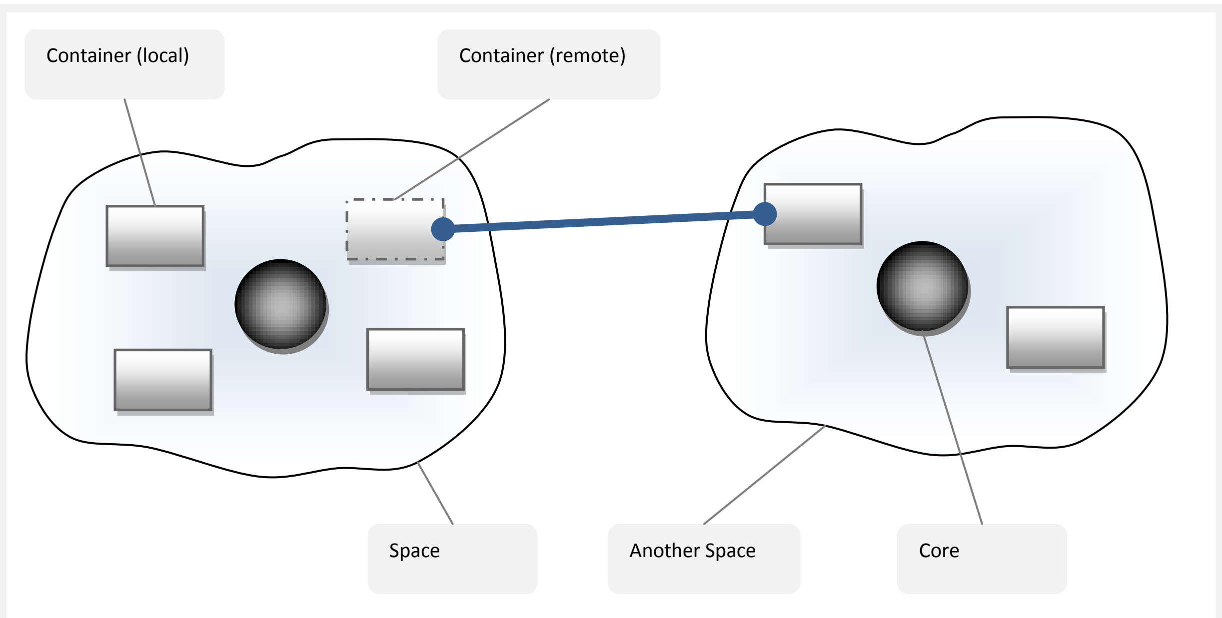


Figure 5: A space with local and remote containers.

3.2 Core Requirements

For the core to be able to provide all the functionality for the space there is a number of requirements that the core must fulfill. First the non-functional requirements [30] are introduced, because many of the functional requirements are partly based on them.

3.2.1 Non-Functional Requirements

The non-functional requirements are particularly important when it comes to designing the core architecture, as the core will be unable to support them when its architecture is not built with having these requirements in mind. Chapter 3.4 describes how these requirements were taken into consideration when designing the core.

3.2.1.1 Concurrency

The core has to be able to handle multiple requests at the same time, coming from both local access and remote communication by another core. The core must be able to handle operations that are independent of each other at the same time, and must coordinate operations that depend on each other (like two operations that access the same container) so that they don't get in in conflict with each other but still run as concurrently as possible. Only in that way it is granted that a space can be used by multiple entities efficiently.

3.2.1.2 Performance

The core is built to be usable for many different usage scenarios, one of its main purposes being *real-time online collaboration* (as introduced in chapter 1). To be able to build "real-time" applications on top of the core, it has to provide the best possible performance, both in local and remote communication.

Also, the core should not only be suitable to run on server machines, but on many different devices, even mobile ones. It is therefore very important that the core is designed as a very light-weight application, using the resources at its disposal in an optimal way.

3.2.1.3 Scalability

The applications which the core is suitable for should range from very simple (e.g. a chat application) to very complex ones (e.g. using the space to coordinate a large number of processes with the need for high performance and concurrency), requiring very low up to high amounts of cpu power. Therefore the core's use of resources has to stay efficient in all kinds of situations, and it has to be able to use more cpu power as it is available (*vertical scalability* [31]), e.g. making use of multiple cpus. It is also important that a network of cores communicating with each other scales well with a growing number of participating cores (*horizontal scalability* [31]).

3.2.1.4 Extensibility

The core is only the base for a large amount of different applications. It is not intended to provide functionality like security features, persistency, replication or many other things you could think of to be very useful in a space. Each of these features would make the core perhaps more suitable for certain scenarios, but would also add unnecessary overhead for many other ones. And this would also clearly counteract the goal of the core being as light-weighted as possible.

Therefore it is crucial for the core to have a maximum support for extensibility. The aim is for extensibility features is to embed them so deeply into the core that it is possible to add even the most complex features (as long as they make sense for a space), like the ones mentioned before (scalability, persistency and replication). The concepts of how new features are added to the core is especially described in chapter 3.6.

3.2.1.5 Stability

The core can only be acceptable as the middleware underlying an application, when the application can undoubtedly rely on it anytime. It is therefore extremely important that no operation can bring the core to an unstable state, and that the core is able to respond to a request even under high load and does not lose any requests at any time.

Of course when used with extensions (like a security add on), these extensions have to provide a certain amount of stability themselves, as they are attached deeply into the core.

3.2.1.6 Fault Tolerance

The core must be as secured against unexpected errors as possible. When an error occurs that doesn't allow an operation to be fulfilled, the core must give back a response with a reasonable error message. This is especially important for core extensions. Whenever a core extension fails to work, the core should if possible recover from the error, or at least be able to give back a reasonable error message, otherwise it would be rather difficult to write reliable extensions for the core.

3.2.1.7 Location Transparency

This is a requirement especially important for distributed systems like the XVSM space. In case of the core, it means that access to containers must not differ, no matter if they are hosted locally or remotely. As soon as a container is known by the core, it is treated in the same way (at least from an outside view), regardless of its location.

However, the core is not intended to provide complete location transparency. The address of a container still has to be known when accessing it for the first time. The lookup and discovery features that are needed to hide the location completely are not part of the core and must be added by extensions, if needed.

3.2.1.8 Documentation

The core will of course try to offer its basic functionality in a way that is as easily understandable as possible. But as a piece of software that can be used in many different ways and that is extensible, understanding especially these extension mechanisms so that they can be used to an optimal degree cannot be a simple task. Wrong usage of the core will as well result in failing to understand the big advantages in using it. It is therefore essential that the core provides a complete and understandable documentation that not only tells the reader what certain core functions do, but also how to use them best.

3.2.2 Functional Requirements

3.2.2.1 Coordinated Data Structures

The most important functionality of the core is to provide coordinated data structures. As already explained above, the data structures in a space are called *containers*. Therefore, a container has to provide possibilities for coordinating its data. "Coordination" in this case means that the data is structured in a certain way that helps users of the space use this container for a certain task. For example, when the container should represent a queue, the data should automatically be structured in a way that always the element that has been written into the container first is also the first one to be read. Since everything that a user does within a space is based on containers, it is essential that the container provides a maximum of performance and reliability. (See chapter 3.3 for a detailed description of containers.)

This feature should also provide extensibility, in a way that types of coordination that are not initially supported by the container can be added by the user him/herself.

3.2.2.2 Notifications

When independent entities are working together on the same data structure (like it is the case with cores in a space), it is important to have the possibility to be informed when changes in this data structure occur in order to avoid polling. Therefore the core must support notifications that notify a space user whenever something has changed in a certain container or if any other operation has

been executed which could be of interest for other users, and also give him/her information about what has been changed. For a detailed explanation of notifications in the XVSM core, see [13].

3.2.2.3 Transaction Support

All operations that can be executed on a container in XVSM (see chapter 3.3) should support the optional use of transactions, similar to a database. The transaction must support commit and rollback. By committing a transaction, all operations that have been done within this transaction are made persistent. By rolling back a transaction, all operations that have been done within this transaction are reverted, the state from before this transaction is restored. Resources (like containers) that are currently in use by a transaction must be reserved for this transaction by locking mechanisms (to prevent two different transactions from conflicting with each other), as long as this transaction is not committed or rolled back. For a detailed explanation of transactions and locking in the XVSM core, see [6].

3.2.2.4 Basic Lookup Functionality

It should be possible to publish a container by a certain name. By knowing this name, together with the address of the core where the container is hosted, a user can access this container. It should as well be possible to unpublish a container that has been published before.

Although this is not a completely location-transparent way of looking up a container (because the address of the core still has to be known), it has been decided not to implement further lookup and discovery services into the core. The reasons for this are that for the cores to find each other can be a very different task according to the situation and can therefore more easily be solved by the overlying application (or a higher level API layer lying between core and application), and that a data structure for lookup/discovery can be very easily managed in the space itself and is therefore not a feature that must be provided by the core directly.

3.2.2.5 Communication between Cores

To be able to access a container that is hosted by a remote core, the core must of course be able to communicate with other cores. The communication process should be completely hidden from the user. As long as a container is known to the user, accessing it should be exactly the same, whether it is hosted by the local core or a remote core. When accessing a container, the local core automatically has to recognize if it has to communicate with a remote core, and know how to reach it. Communication between cores should be connectionless (at least from an outside view), meaning the user never has to deal with opening or closing remote connections.

The communication functionality should also support extensibility, by providing the possibility to completely replace the current communication service by a user-implemented one. This is to enable the user to write communication services for special usage situations, e.g. a communication service that is specialized on high performance communication over LAN only, or a communication service specialized for overcoming firewalls. (The default communication service provided by the core should of course be one that is suitable for most usage situations.) For further introduction of the remote communication mechanisms in the core see chapter 3.4.3.

3.2.2.6 Aspects

The most powerful extensibility feature is the support of *aspects*. An aspect is a piece of code that can be inserted at certain points within the core. It should be possible to add aspects to any

operation that can be done with the core (to be executed directly before or after the operation) and thereby alter the operation's behavior, including reading from and writing to containers, creating and destroying containers, and creating, committing and rolling back transactions.

It should be possible to add aspects to the whole (local) space, meaning that the aspect is executed every time when the operation where the aspect has been added is executed. This is called a *global aspect* or *space aspect*. On the other hand, an aspect could also only be added to the operation of a single container (e.g. it is executed every time something is written into the container). This is called a *local aspect* or *container aspect*.

This feature must be powerful enough to implement every thinkable extension that could be of use for the core. This includes security, persistency and replication features, and many more. For a detailed explanation of aspects in the XVSM core, see [13].

3.2.2.7 Low Level API

The core must provide an API that supports all of the core's functionality, like container and transaction operations. This API is more aimed for supporting the complete feature set of the core than providing maximum usability, which would be difficult to implement, as there are many features (above all the extensibility features) that require a rather detailed knowledge of the space's internals, but will not be needed by most users. Therefore, providing a maximum of usability is left to higher level APIs that specialize on a certain subset of core operations. This feature is further introduced in chapter 3.4.2.

3.2.3 Non-Core Functions

It has already been mentioned that the core is aimed to be light-weight and support maximum extensibility. For the core to really be as light-weight as possible, it is essential to reduce the feature set to an absolute minimum, but also to still be useful guarantee that important features that are not part of the core can be added later by extensions.

It will of course normally be easier to make a feature best optimized for reliability and performance when it is implemented right into the core itself. Because of that it is for some features not an easy decision if they should be part of the core and therefore make it heavier and perhaps also more susceptible to errors, but also make the core more useful because the features could be important in many situations.

For some features it has already been explained above why they were not taken into the core feature set. There are some more features (that have also already been mentioned above in short) for which the reasons for taking them not into the core should be explained. (For more examples of extension features of the core, take a look at chapter 3.6 which deals with *profiles*.)

3.2.3.1 Security

Security is very important for middleware systems, and in particular for space. It is crucial for users of the space to be able to *authenticate* themselves when accessing data in the space, and to get *authorization* for accessing certain data. An important security feature is also *encryption* when remote communication is involved.

All of these features are clearly needed as soon as it comes to coordinating sensitive data and preventing misuse of the space, and would speak for incorporating security features into the core.

On the other hand, security is also clearly a very heavy feature. But the main reason for not implementing security is the wide variety of possibilities how security could be implemented (or would be expected from a space). There can be very different answers to questions like: How should users be managed in the space? Which encryption modes should be supported? Which access rights need to be defined? Implementing a certain security model to satisfy most possible usage situations would be a difficult task. Because of that, it was decided that security features would be left to implement on top of the core by the use of aspects.

3.2.3.2 Persistency

The possibility of persisting the data within a space (just like in a database) clearly adds a lot of functionality to the space. It would make the space a lot more fault tolerant, because if a system crashes, the space could recover its own data after restarting and nothing would be lost.

But it has to be kept in mind that the primary goal of the space is to coordinate, and not to store data (which is the clear difference between a space and a database). Of course persistency would be a benefit in many coordination scenarios as well, but it is not that important as to take a heavy feature like persistency into the core. Aspects should enable users to add persistency to the core in exactly the way they like, e.g. using a database underneath the core to store the space's data.

3.2.3.3 Replication

In distributed systems another important feature is replication. When data is replicated to different cores, this adds a great deal of availability, because even when one core becomes unavailable for a certain amount of time, the data can still be accessed from the other cores that store replicas of it. Also, to compete with other distributed systems (like GigaSpaces) on the long run, it is clear that the possibility of adding replication will be needed.

But replication is also a very complex issue. It is a feature that is initially not needed by most usage scenarios, and is rather used more to add stability and availability to certain scenarios. Therefore it was also in this case decided to enable aspects to implement replication features, and leave this feature out of the core.

3.3 Containers

The space's goal of providing coordinated data structures has already been addressed in short. The following chapter will give more detailed information about how these data structures look like.

Within a space, a *container* is a structured data storage that can contain *entries*. It must not only be flexible in the way data is coordinated and support all kinds of different data types, but must also support transactions and deal with concurrent operations, e.g. when one user wants to read something from the container and another one wants to write something into the same container at the same time.

In general, a container can be imagined as shown in Figure 6. This graphical notation will from now on be used for a container throughout this thesis.

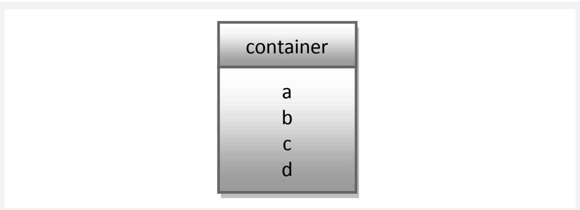


Figure 6: A container holding the elements a, b, c and d.

To be able to use the container for coordinating data in every thinkable usage situation, it has to be highly adaptable and allow a wide variety of different types of coordination. In respect to that, a container's coordination type(s) can be seen as the most important property of a container.

3.3.1 Coordination Type

A *coordination type* defines for a container how its entries are coordinated. It defines for example a certain order for the entries in a container, or which entries are read next in a read operation. All coordination types of a container have to be defined at the time the container is created.

Possible coordination types are for example *fifo* (elements ordered like in a queue), *lifo* (like a stack), *vector* (like a list) and *key* (like a hashtable/dictionary). Figure 7 shows a *fifo* coordinated container, with *a* being the entry that has been written first (and will also be the one to be read first) and *d* written last. For further explanation, see the chapter about coordination types in [12].

container	fifo order
a	1 (first)
b	2
c	3
d	4 (last)

Figure 7: A Container that is coordinated by fifo coordination type.

3.3.2 Selector

Selector is a general term for selecting the entries that are read from a container, or specifying how entries are written into the container. A *read selector* specifies the coordination information (order, key values, vector indices, …) and count of the entries that should be read from a container. A *write selector* specifies the coordination information (order, key values, vector indices, …) for entries that should be written into the container. For every available coordination type there is a certain selector that comes to use always when operations are done on a container that is coordinated by this coordination type.

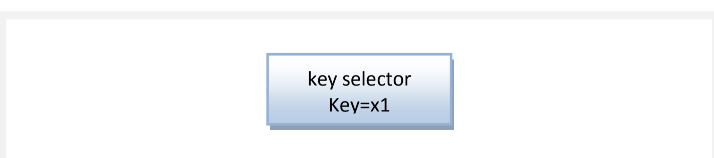


Figure 8: A Selector for the coordination type *key* that is used to read / write an element with key "x1".

3.3.3 Entry

Having defined the meaning of coordination types and selectors, it can now be defined what exactly an entry is. An entry consists of:

- a value (the data to be stored),
- information about the type of value (like a string, tuple or other object)
- and the selectors that belong to this value (like a key value or vector index).

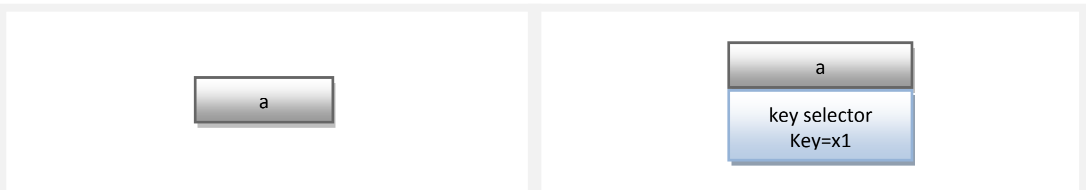


Figure 9: An entry with value "a" and no selectors.

Figure 10: An entry with value "a" and a selector that defines the key value "x1" for this entry, for a *key* coordinated container.

3.3.4 Container Properties

The container has several properties that specify its behavior and that have to be defined by the time the container is created.

3.3.4.1 Size

- **Bounded**

  A *bounded* container has a specific size that defines how many entries it can hold. For example, a container with a size of 4 can hold 4 entries a most.

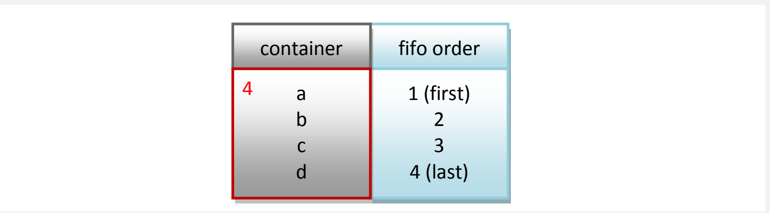


Figure 11: A container that is bounded to size 4. Because there are 4 entries in it, the container is full and cannot take any more Entries.

- **Unbounded**

  An *unbounded* container has no size limit, and therefore can hold an unlimited number of entries.

3.3.4.2 Uniqueness

In a container with uniqueness constraints, no two entries can exist with the same value. For example, it would not be possible to write an entry into the container with value "a", if another entry with value "a" already exists in this container. A container without uniqueness constraints doesn't have this restriction.

3.3.4.3 Coordination Types

The most important property of a container is its coordination types. As already described above, the coordination types of a container describe how the entries within this container are coordinated. A container can have multiple coordination types (and must have at least one, a container without any coordination would not be useable).

3.3.5 Container Operations

The operations of a container can be split into read and write operations. For a detailed explanation how container operations behave in combination with certain coordination types, see the chapter about coordination types in[12].

3.3.5.1 Read Operations

A read operation reads entries from the specified container. It uses selectors to specify which entries should be read and the number of entries that should be read:

- In the selector a count can be defined that says how many entries should be read from the container. If a count is defined, the read operation will block until the defined number of entries is available. If no count is specified, the read operation doesn't block and immediately returns all available entries that apply to the selector.
- If a count is specified that is greater than the maximum number of entries in a bounded container, an error will be thrown.

There are three different types of read operations:

- **Read**

  The standard operation is *read*, where the entries are simply read from the container, the container itself is not changed. Figure 12 shows the process of reading an entry from a *fifo* coordinated container. The entry is returned and the container is not changed.

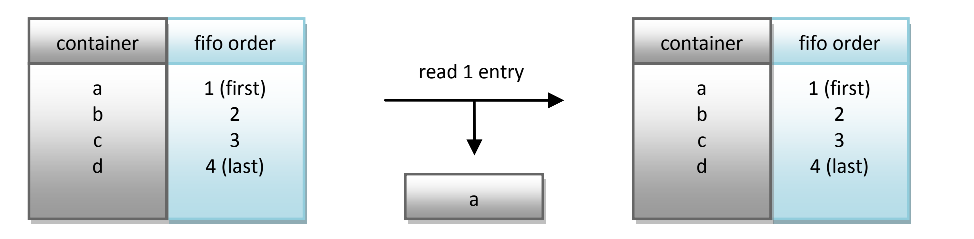


Figure 12: Reading an entry from a *fifo* coordinated container.

- **Take**

  If *take* is used as type, all the entries that are read are removed from the container. Figure 13 shows how an entry is taken from a container. The entry is returned and removed from the container.

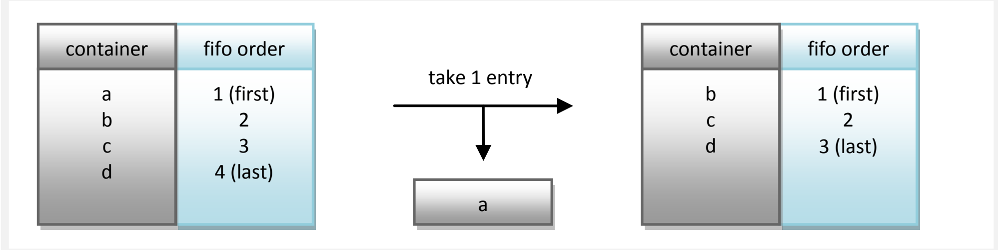


Figure 13: Taking an entry from a *fifo* coordinated container.

- **Destroy**

  Destroy removes all the entries that are read from the container like *take*, but doesn't give the entries back. Just an "acknowledged" is given back if the operation was successful. Figure 14 illustrates this.

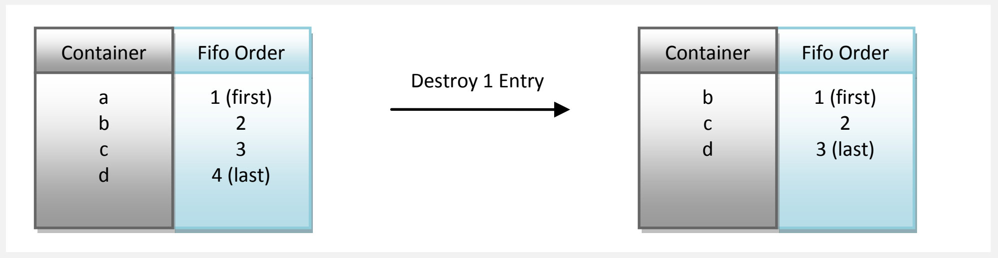


Figure 14: Destroying an entry in a container with *fifo* coordination.

3.3.5.2 Write Operations

A write operation writes entries into a specified container. A single write operation can contain one or more entries. If the operation consists of more than one entry, the entries are only written into the container if all entries can be written successfully. The selectors of the entries define how each entry will be coordinated in the container.

There are two different types of write operations:

- **Write**
  If a container is bounded and has reached its maximum size, or if there are any entries in the container that would prevent the new entry from being written (e.g. in a *key* coordinated container, when a key value already exists), a *write* operation blocks until there is enough space in the container.

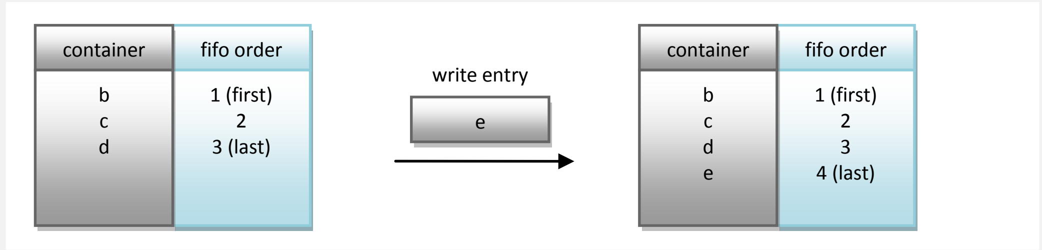


Figure 15: An entry is written to a container with *fifo* coordination.

- **Shift**
  A *shift* operation never blocks (except in situation where a container or entry is locked, see chapter 3.3.6). Its purpose is to be used when an entry should absolutely be written into a container, even if this means that other entries in the container would need to be removed. If there is not enough space for an entry to be written into the container (in case the container is bounded), another entry is removed from the container (see Figure 17). Which entry is removed from the container is decided on basis of the container's coordination types (this subject is handled in detail in [12]).
  Also, if there are any entries in the container that would prevent the new entry from being written (e.g. in a *key* coordinated container, when an entry with the same key value already exists, see Figure 16), these entries are removed from the container (in case of *key* coordination, you could also say the new entry overwrites the old one).

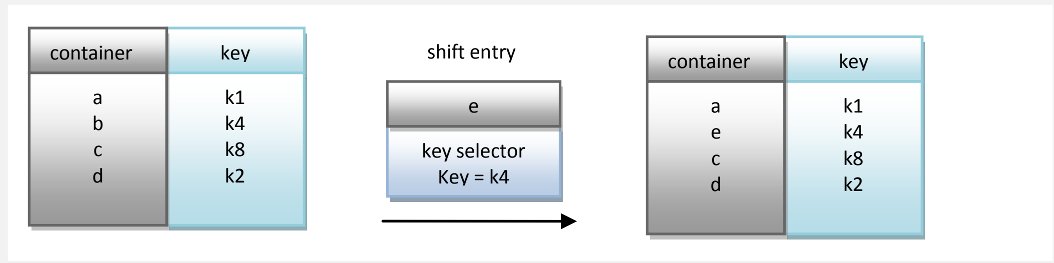


Figure 16: A new entry is shifted into a *key* coordinated container and replaces another one.

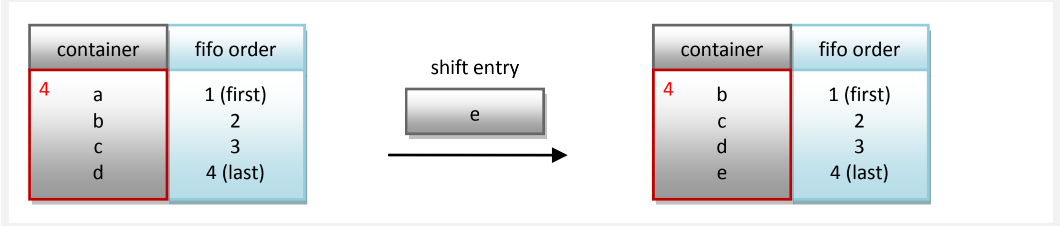


Figure 17: An entry is shifted into a bounded container that is already full, which causes another entry (a) to be removed.

3.3.6 Transactions and Concurrency

The support for concurrency is very important for a container. To qualify as a coordinated data structure, the container has to guarantee that it will coordinate different users accessing the same data at the same time with the highest possible concurrency, but at the same time prevent inconsistencies, e.g. it should be possible for different users to read the same entry at the same time, but it must not be possible to take the same entry (using the *take* operation) at the same time (because this would mean that the same entry has been taken twice, which must of course not be possible).

This issue is directly connected to the support for transactions. The container must be able to support the use of transactions, which means that when an operation like *take* or *write* is done using a transaction, it will only take effect as soon as the transaction is committed, or be completely undone if the transaction is rolled back. But as long as the transaction is still active, no other operation that would offend the outcome of the first one can be executed. For example, if an entry is taken from a *fifo* coordinated container within a transaction, no other operation (despite one within the same transaction of course) can take an entry from this container, because it would take the same entry as the one already taken by the other operation (and taking the next one would hurt the rules of *fifo* coordination, because as long as the transaction is not committed, the entry that has been taken is actually still part of the container). In other words it means that as long as the transaction has not been committed or rolled back the container (or at least the affected entry, the level of locking depends on which coordination type is used) is locked.

Therefore, locking is a very important mechanism concerning transactions and concurrency. If an operation wants to access a resource (container or entry) that is locked by a foreign transaction (or the operation doesn't explicitly use a transaction at all, because even in that case it will need to get a lock for the resource for a very short amount of time), it will block until the lock is removed from this resource (or to be more precise, until all locks are removed that prevent the operation from executing). An operation will wait until it has reached its timeout.

The container does not specify which locking behavior is used exactly, this has to be decided by the implementation. But the most recommended approach is to follow the multiple-reader/single-writer principle, which means that multiple readers can access a resource at the same time, but only one writer at a time. If one transaction has written something to a resource, no other transaction can read from or write to this resource until the transaction that has written is committed or rolled back. On the other side, if a transaction has read from the resource, other transactions can still read as well, but no transaction will be able to write until all transactions that have read from the resource are committed or rolled back. Note that reader/writer in the sense of locking doesn't mean exactly the same thing as the read/write operations of the container. A read lock is acquired only when the operation doesn't change anything in the container, which in only the case with the *read* operation. All other operations (*take*, *destroy*, *write*, *shift*) need a write lock.

For more information about transactions and locking, see [6].

3.3.7 Meta Container

Metadata is information about data. It is used to facilitate the understanding, use and management of data. In case of a container, metadata would be general information about the container itself that does not belong to a certain entry within the container. Because such information can be very useful, every container in a space has its own so called *meta container*. A meta container is nothing else than a *key* coordinated container that can store key-value pairs (with a string as key).

Metadata to be stored for a container is:

- The name of the container
- A description for the container
- The maximum size of the container
- The container's current entry count
- The creation date of the container
- The name of the container's creator
- Relationships to other containers in the space, like one that contains a log about this container's activities
- Additionally a user could store own data in the meta container, which could be of any type and with any content

As seen in Figure 18, the data in the meta container can be divided into two different types:

- **User-defined data**
  Information that has been added to the meta container by users of the space and can be changed by them. The space user can define the key-pair s/he wants to add by her/himself, except that s/he cannot choose a key name that is already present.
- **Non-user-defined data**
  Information that is added to every meta container by the time its container is created. This information could either be read-only, like for example the creation date of the Container, or could also be changeable by space users, like a container description that is by default empty at creation time and can later be set to a fitting text.

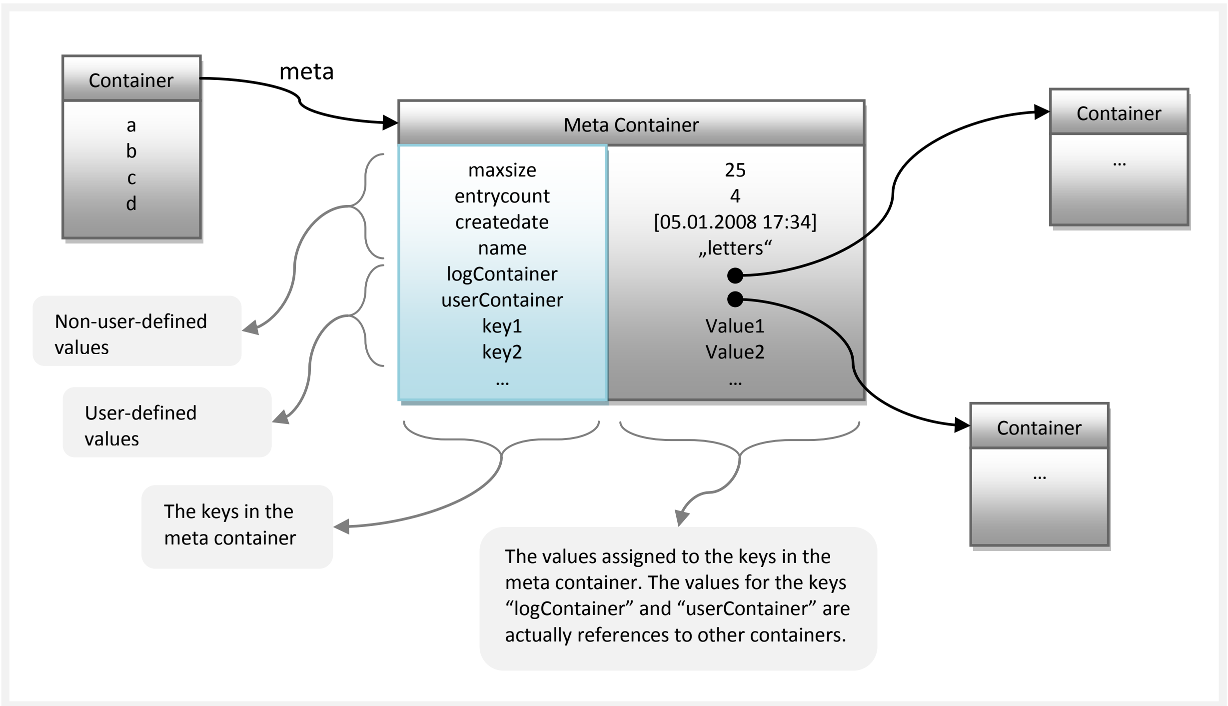


Figure 18: A container and its meta container, containing several user-defined and non-user-defined values.

There are only two (non-user defined) values that are always part of the meta container, namely the *maximum size* of the container and the *current entry count* (both of them are read only). While the maximum size is a constant value that is never changed, the current entry count is managed by the container and updated every time an entry is added to or removed from the container.

3.4 The Core Architecture

After knowing the most important requirements for the core and understanding the structure and functionality of containers, the structure of the core itself has to be defined. As parts of the main requirements are performance and concurrency, it is clear that the core must be built as a multithreaded system. The core follows the staged event-driven architecture [32] approach, which not only supports building a concurrent multithreaded system, but also helps creating a very clear structure because the single components are very loosely coupled.

Figure 19 shows the basic structure of the core. To allow each component to run on separated threads, communication between components is not done directly by method calls. Containers, similar to the ones used in a space, are used for thread-independent message passing. To distinguish them from the containers in the space, they are further called *core containers* (colored gray in the figure; core containers are further described in chapter 3.4.1). For all situations in the core when two components must communicate with each other, they use a core container to do that. In that way the components completely keep their independence. There are three different types of messages that are passed between components, marked with three different colors. Request messages (blue) contain information that has yet to be processed by the core. Response messages (green) contain answers to request messages, every request message entering the core leads to a response message leaving it. The third type of message is the event message (red). It is used only inside the core, to wake up waiting request messages. The use of events is explained later in detail.

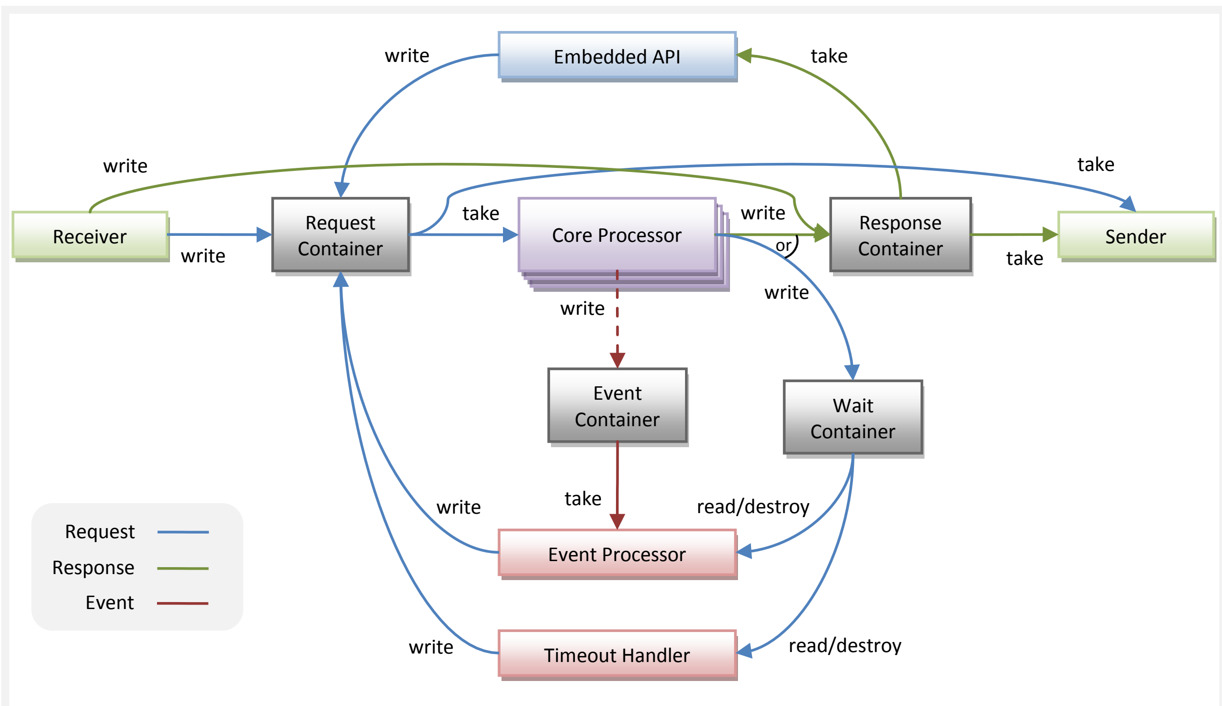


Figure 19: Basic core structure.

The standard scenario is as follows: An application communicates with the local core by using the *embedded API* (described in chapter 3.4.2). Every time a method in the embedded API is called (e.g. the method for creating a container) a request message is generated. This message is written into the *request container*, from where it is taken by the *core processor* (handled in chapter 3.4.4) that does whatever is needed to fulfill the requested operation (e.g. creating the requested container). The core processor then creates a response fitting the operation that has been done (e.g. when a container is created, the content of the response message would be a reference to the container) and writes it to the *response container*, from where it is taken by the embedded API that uses the information contained in the response and returns it to the calling application.

If the request would have been made not locally, but by a remote core, it would have been received by remote communication with the *receiver* component that again writes the request into the request container. The processing works the same as above, only that when the response message is written into the response container, it is taken by the *sender* and sent to its remote destination. These components are described in chapter 3.4.3.

3.4.1 Core Containers

The core containers are used for communication between the components of the core. To fully understand the structure of the core, it is first important to understand how the core containers are used. Similar to a container in a space, a core container provides several operations for reading and writing data:

- **write**
  Writes a single entry (request, response or event) into the container.
- **read**
  Reads a single entry from the container. If the container is empty, the operation blocks until an entry is written to the container that can be read. The entry that is read next is always the

one that has been written to the container first (like in a container with *fifo* coordination), in other words the core container is basically ordered like in a queue.

- **take**
  Takes a single entry from the container. This works exactly like read, only that the read entry is also removed from the container.
- **destroy**
  Removes a certain entry from the container, if this entry is in the container.

Also similar to the container in a space, the core container must be absolutely capable of handling concurrent calls (read and/or write operations at the same time). It is also very important that the core container in an XVSM implementation is built for maximum performance, else it would build a bottleneck for both the requests entering the core and the responses leaving it. Differences of the core container from a container in the space are that it doesn't support transactions and has no extensibility for additional types of coordination (because the built in coordination described in the operations above is absolutely sufficient).

There are four core containers in the core with different tasks:

3.4.1.1 Request Container

The request container receives all requests that should be processed. It is therefore read by the core processor that has the main target of processing all requests that are received by the core. The core processor is actually more thought of as a pool of processors, being able to process many requests concurrently if needed (see chapter 3.4.4). Because the core containers support concurrency, all processors can take requests from the request container at the same time without problems. If there are no requests in the container, the processors will wait with a blocking take call, and as soon as a request is written into the container, one of the blocking takes will be woken up and return the newly written request to the processor.

The figure shows that not only the core processor takes requests from the request container, but also the sender. This is because in the case when a request cannot be fulfilled by the local core, e.g. when it deals with a container hosted by a remote core, it has to be sent to that core so that it can be processed there, so it is important that local requests can be distinguished from remote requests. Because of that, elements in the request container are additionally flagged by a "local" or "remote" flag. When doing a take operation on the request container with using one of these flags, only the responses with this flag are returned. Because of that it is possible for the core processor and the sender to do a take operation and still only receive what they want, by the core processor using the "local" flag and the sender using the "remote" flag.

3.4.1.2 Response Container

All responses that are generated for successfully processed requests by the core processor are written into the response container. Similar to the request container, the response container has two different components that take responses from it, because local responses need to be taken by the embedded API and remote responses need to be taken by the sender. Because of that, the response container also uses the "local" and "remote" flags for distributing the responses to the correct components. The embedded API takes all local responses while the sender deals with all remote responses.

3.4.1.3 Wait Container

It has been explained earlier in the chapter about containers that in some situations container operations are delayed, like when trying to read an entry from an empty container, or when the container is currently locked by another transaction. This waiting mechanism also has to be implemented in the core.

Therefore the core has a wait container. Whenever a request cannot be processed immediately (e.g. a take operation cannot be processed because the container is currently empty, see chapter 3.4.5), it is written to the wait container. As soon as an event occurs in the core (e.g. an event would be that something has been written into a container), the *event processor* component checks the wait container if there are any requests that should be woken up because of that (e.g. request waiting for entries to be written into that container), removes these requests from the wait container and puts them back into the request container to be processed again. Requests that reach their timeout before they are processed again are removed from the wait container by the *timeout handler* component. The event processor and timeout handler components are explained in detail in chapter 3.4.5.

3.4.1.4 Event Container

This container receives events by the core processor. The events are read by the event processor to check if there are any waiting requests that can be woken up because of this event. An event contains all information needed to decide if a certain request should be woken up or not, for example a reference to the affected container and the information if entries have been added to or removed from this container. Event processing is explained in detail in chapter 3.4.5.

3.4.2 The Embedded API

The embedded API allows applications to communicate locally with the core. It contains methods for every operation that can be done at space level, like creating and destroying containers, creating, committing and rolling back transactions, and all container operations for reading and writing. Every time a method in the embedded API is called a request message is generated. This message is written into the request container. After that the embedded API waits until a message is written into the response container that is the response to that request. Every request message is given a unique id by the embedded API, and a response is recognized as belonging to a certain request when it has the same id (the core processor takes care that every response is provided with the id of the belonging request).

Figure 20 shows the standard scenario for communication with the core that has also already been explained above in short. The request shown here is one for creating a container. It therefore must contain all the information needed so that the container can be created (container size, coordination types, …). The message also contains a unique id and is flagged “local” to correctly be recognized as a request that has to be processed by the local core. After the core processor has finished processing the request (which in this case means it has created the requested container), it generates a response message and provides it with the id of the request. The response message is again flagged with “local” so that it is taken from the response container by the embedded API. The response contains any information that is important for the action that has been done, in case of container creation it contains the reference to the created container (the container’s address that is used to access it).

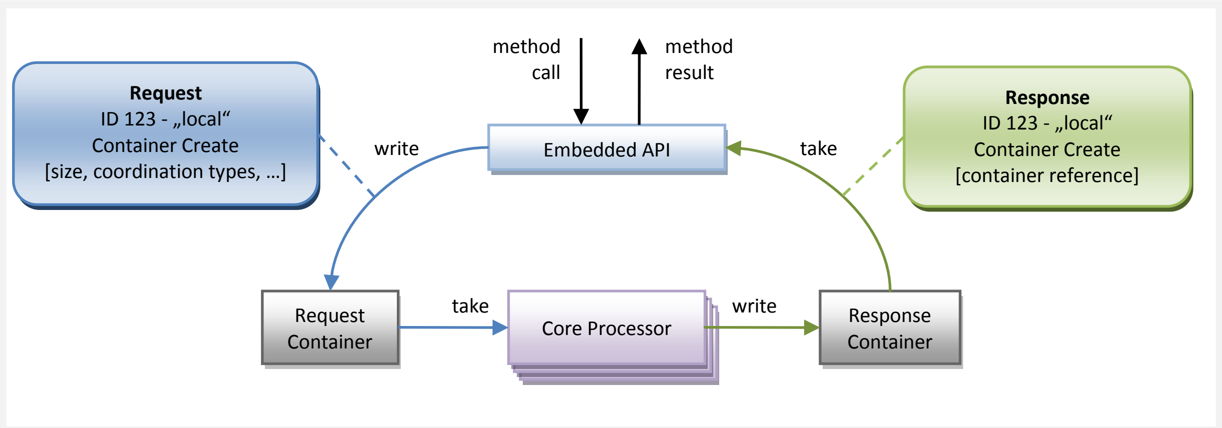


Figure 20: Standard scenario of a local method call in the embedded API.

How the API exactly looks like is a question of the implementation, the model doesn't specify it in detail. It could for example provide methods for either synchronous or asynchronous core access, or for both. Synchronous core access would look just like a simple method call. Within the API the call would block until the response to the request that has been given to the core is received, and then return the content of the response to the user as a result of the called method. Using asynchronous access, the API could give the user the possibility to define a callback method, and as soon as the response is received that method is called. This is also called the *result callback* pattern. Other useful patterns that could be implemented for asynchronous communication would be *fire and forget*, *sync with server* or *poll object*, see [33].

3.4.3 Remote Communication

The *sender* and *receiver* components are the parts of the core that handle remote communication with other cores. They are responsible for sending requests that have to be handled by remote cores and receiving responses to requests that were handled by remote cores, as well as receiving requests from remote cores that need to be processed by the local core and sending responses to such requests back to them.

Figure 21 shows the scenario for remote communication between two cores. Like above, a container should be created, but this time not at the local core (core A), but at a remote core (core B). In this case the caller of the method in the embedded API has to know the address of the remote core where the container should be created. For simplicity reasons we assume that the address of core A is simply *CoreA* and the one of core B is *CoreB*. The figure shows that the request created by the embedded API is not only flagged with "remote" so that it will be recognized as having to be sent to another core, it also contains the address of the remote core it has to be sent to (*CoreB*). The sender at core A takes the request from the request container and sends it to core B. Additionally the sender adds the own core's address to the request, so the other core knows where it comes from. On the other side the request is received by the receiver, now including the address *CoreA*. The receiver writes the request to the request Container, flagged with "local" because it should be processed by the local core.

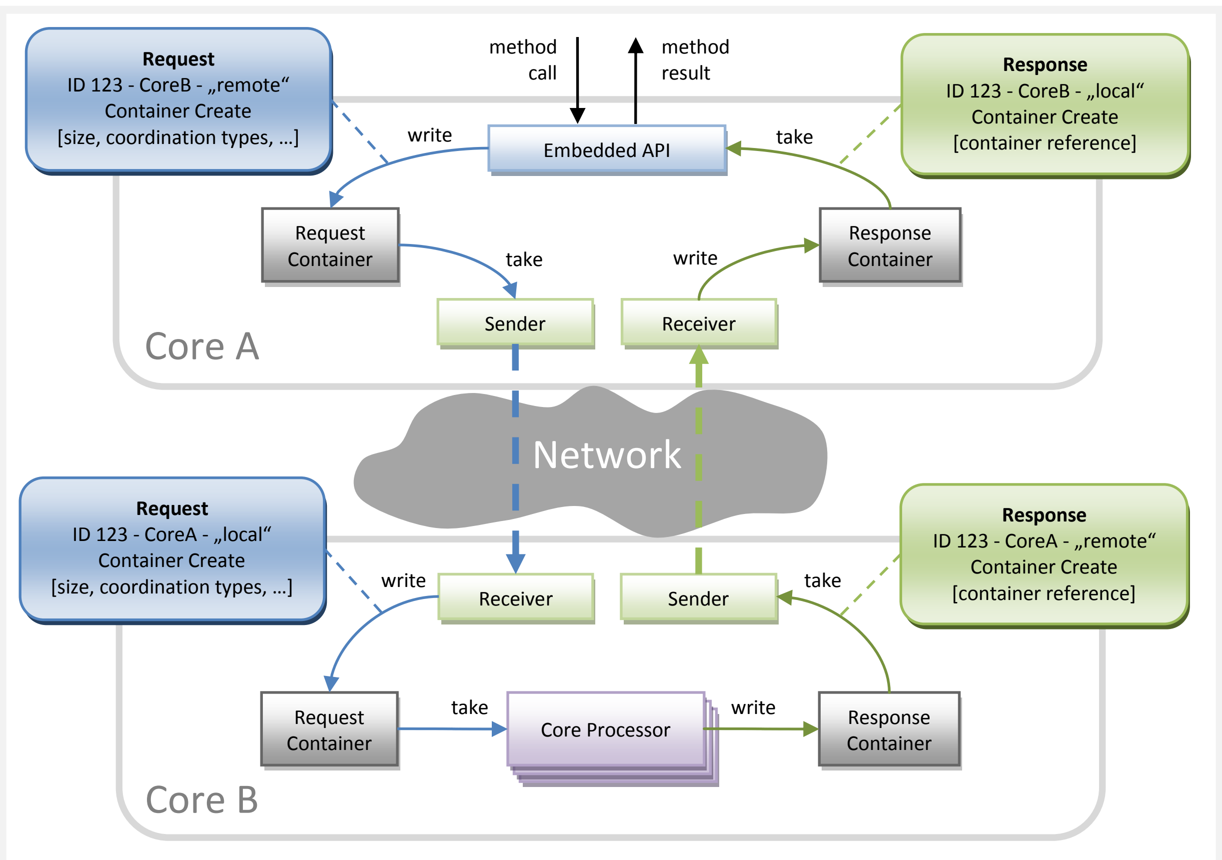


Figure 21: Standard scenario for remote communication between two cores.

After processing (which works the same no matter if the message comes from the local or a remote core) the core processor writes the response into the response container. Because the address contained in the request is not the one of the local core, the response it flagged with “remote”. It is then taken by the sender and sent to the given address, *CoreA*. Again the sender adds the own core’s address to the message. Back in core A, the response is written to the response container flagged “local” so that it is taken by the embedded API.

This may seem a bit complicated at the first look, but it isn’t. On the contrary, it helps keeping all components separated, and it also helps keeping a clear and simple design for sender and receiver. The tasks for sender and receiver can now easily be summed up: The sender takes messages flagged with “remote” from the request and response containers. It adds the local core’s address to the message and sends the message to the remote core’s address contained in it. The receiver waits for incoming messages from remote cores. When a messages is received, it is determined if the message is a request or a response. Requests are written into the request container, responses into the response container, both flagged with “local” (because the received requests are always to be processed in the local core, and the received responses are always meant for the embedded API).

The clear definition of the sender’s and receiver’s tasks is also important because of the needed support for extensibility. The communication components must be completely (and easily) replaceable regarding transport protocol (like TCP or UDP) and encoding protocol (like XML or binary) so that the user is able to replace the standard communication components by his/her own ones that can be specialized for certain scenarios. Further details about remote communication mechanisms in the XVSM core are handled in [12].

3.4.4 The Core Processor

The core processor is the most important component of the core. It implements nearly all of the core's functional requirements, like creating and destroying containers, creating, committing and rolling back transactions, and all the container operations for reading and writing. In other words, the core processor has the functionality to process every incoming request and create an appropriate response.

The core processor actually consists of a pool of processors that are able to process requests concurrently (and all have the same functionality). This is why in all figures the core processor is shown not as a single box, but as a stack of boxes. In addition, the core processor can be split into several smaller components, which also helps keeping this large component less complex.

The structure of the core processor is shown in Figure 22. It is based upon the idea to split the operational components from the data structure components. This not only helps reducing complexity, but is also very helpful for coordinated access of the core's data, because although multiple processors may be able to process requests concurrently, when more than one processor tries to access the same resource (like a container) their access has to be coordinated.

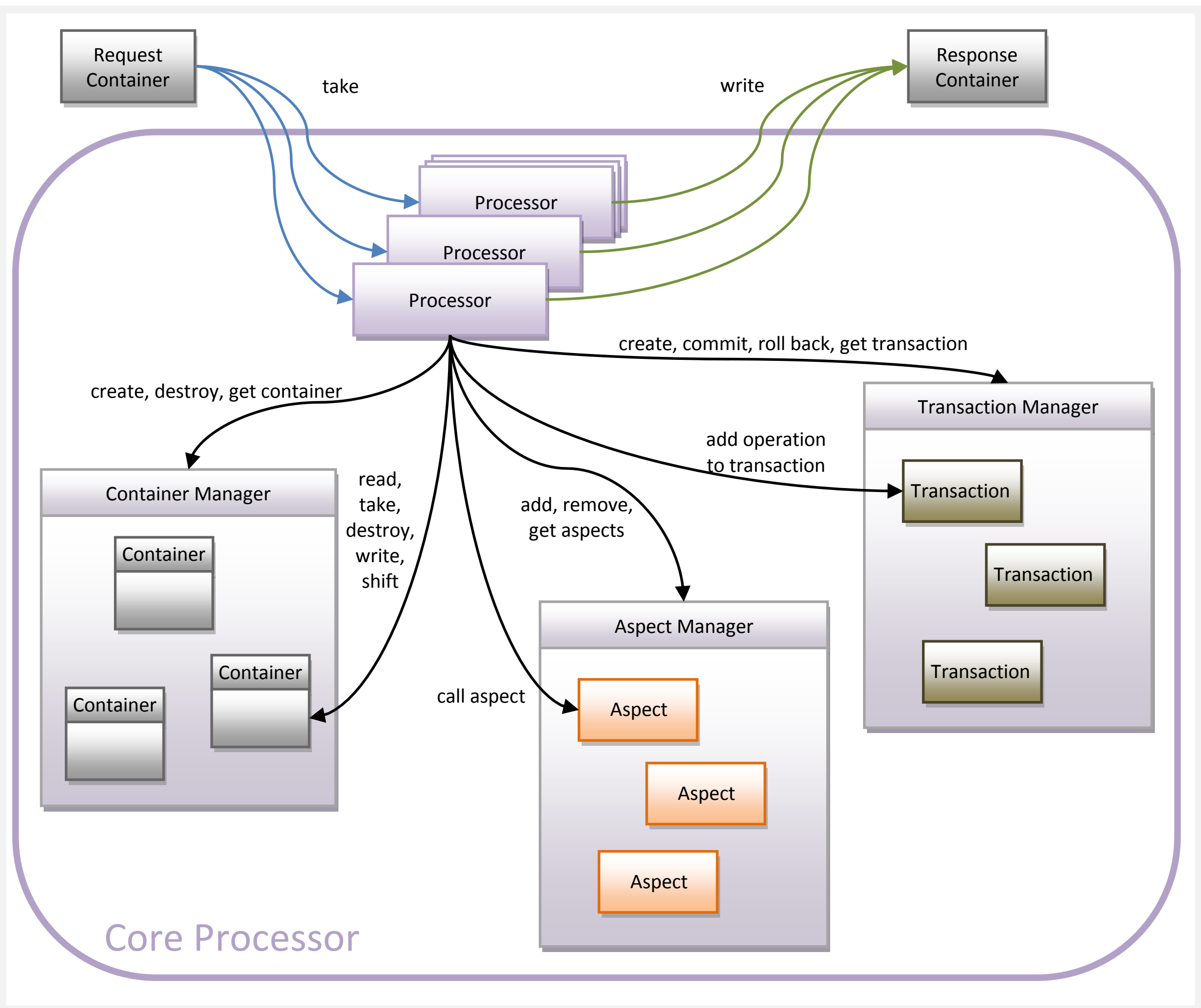


Figure 22: Structure of the core processor.

3.4.4.1 Processor Pool

The first thing shown in Figure 22 is the pool of processors for processing requests. Every processor is doing a take operation on the request container (using the "local" flag). As soon as a processor receives a request, it processes that request, creates a response and writes it into the response container (at least in the standard case; we will deal with waiting requests later). Afterwards, it takes the next request from the request container. The number of processors can vary, since it could depend on the system power and usage scenarios how many processors are best to be used, and can therefore be set depending on the usage situation.

3.4.4.2 Container Manager

The second core processor component is the container manager. It manages all the containers that are hosted by the core. It provides thread safe methods for creating, destroying and retrieving containers. So for example when creating a container, a processor just calls the appropriate method of the container manager and does not need to handle any concurrency problems, because even if more than one processor wants to create a container at the same time, the container manager will coordinate these calls correctly.

The data structures that are hosted by the container manager, representing the core's containers, provide thread safe methods for the operations read, take, destroy, write and shift for a certain container. If a processor wants to process a write request, it first has to retrieve the container from the container manager to which the data should be written, and then use the write operation of the retrieved container data structure. Because the methods are thread safe, the processor doesn't have to deal with concurrency issues, the container data structure will coordinate concurrent access from more than one processor itself.

3.4.4.3 Transaction Manager

The third component is the transaction manager. It manages all the transactions that are currently active in the core. The transaction manager is somewhat similar to the container manager, it provides thread safe methods for creating, retrieving, committing and rolling back transactions. The transaction data structures stored in it provide thread safe methods for adding operations to this transaction. A single transaction stores all its operations in the order they were executed.

3.4.4.4 Aspect Manager

The last component is the aspect manager. This component is essential for the support of aspects. As described earlier when dealing with the functional requirements of the core, an aspect is a piece of code that is added to the core at a certain point. Such a point where aspects can be inserted is called *insertion point*, or in short *ipoint*. Every operation in the core has two ipoints, one directly before (the *pre* ipoint) and another directly after (the *post* ipoint) the operation's execution. Because the core processor is responsible for executing all requested operations, it is also the best place for implementing the support of aspects. The aspect manager keeps a list of aspects for every insertion point. Like the other manager components it provides thread safe methods for adding and removing aspects and for getting the list of aspects of a certain insertion point. Every aspect stored by the aspect manager provides a method that executes the aspect code when it is called. Since an aspect is not initially part of the core itself but is written and added by a user of the core, the creator of the aspect has to provide thread-safety for resources that are used by the aspect her/himself if needed.

Every time a request is processed by a processor, the processor first gets the list of aspects for the pre ipoint of the operation being processed from the aspect manager. Then the processor executes all aspects in the list (or none, when there are no aspects for this ipoint). After that the operation itself is executed. Directly after the execution of the operation is finished, the processor loads the list of aspects from the aspect manager for the operation's post ipoint, and executes all aspects in the list. Aspects can also have special return values that for example tell the processor to skip an operation completely. Further details about aspects in the XVSM core are handled in [13].

3.4.5 Event Processing and Timeout Handling

One thing that the core processor doesn't handle directly is blocking and handling timeouts of unfulfillable requests (as said in earlier chapters, requests need to be delayed for example when there are not enough entries in the container to be read, or when another transaction currently blocks the container that should be accessed). If the core processor would do a direct form of blocking wait, it would mean to entirely block one processor thread for the time it takes for the request for be fulfilled (or until it times out). In a worst-case scenario, when the core receives a large amount of requests that cannot be fulfilled it would block the entire processor pool and thereby make the core unable to process any more requests (assuming that the number of processors is limited to a certain size, which would normally be the case).

Therefore the core implements a system for handling blocking requests without the need of blocking any of the core's operational components. The main components of this system are the *wait container*, the *event container*, the *event processor* and the *timeout handler*. As soon as a request cannot directly be fulfilled and therefore has to be delayed, the core processor doesn't generate a response to the request but instead writes the request into the wait container.

3.4.5.1 Event Processing

Figure 23 shows the scenario when processing an unfulfillable request, in this case a read operation to a container that is empty, which means that the read operation is delayed until entries are added to the container, or until it runs into a timeout. All requests for operations accessing a container have to provide a certain timeout value, in this case this value is 1000 milliseconds (configurable for every operation). First the request is taken from the request container by the core processor (1). Because the request cannot be fulfilled, the core processor writes it into the wait container (2). When now something is written into the same container, the core processor creates an event and writes it into the event container. This event contains information about what happened that could be important for waiting requests, in this case the information that entries have been added to a certain container (3). The line in the figure going to the event container is dashed because not every request that is processed by the core leads to the creation of an event. An event only has to be created when the operation that is processed has a chance to wake up a delayed request. For example, creating a transaction could never lead to waking up another request, and so will never create any event.

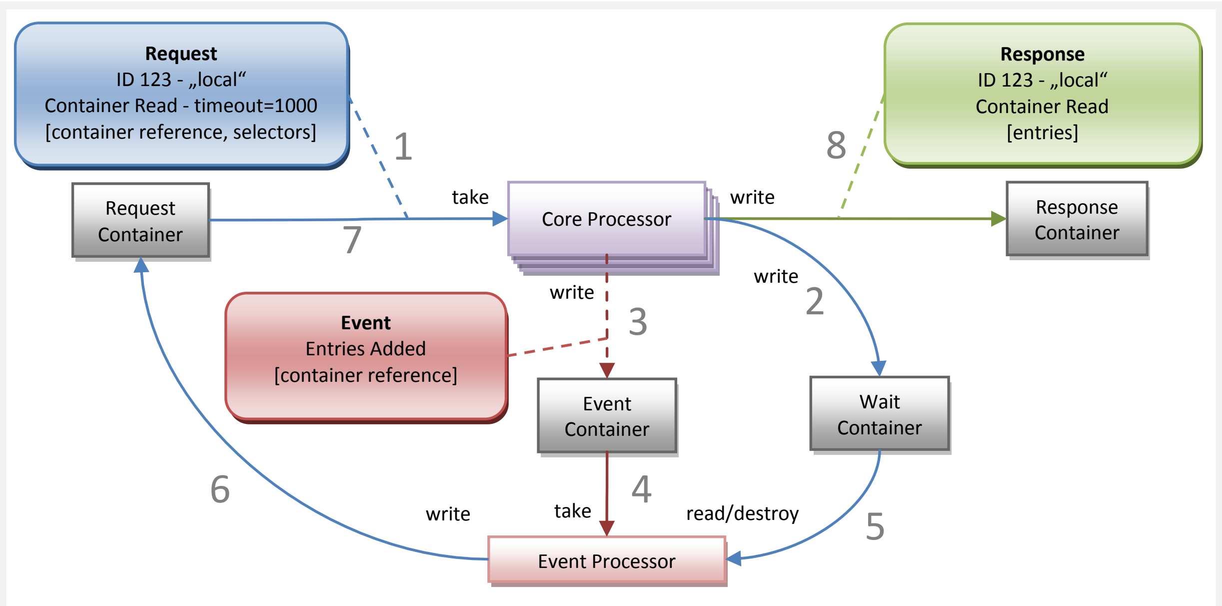


Figure 23: The core wait and event processing structure.

After the event has been written into the event container, it is taken by the event processor (4). The event processor has the task of waking up requests that are waiting for this event. As soon as the event processor gets an event, it reads all requests from the wait container and checks for every request if the event is a reason to wake it up (5). In this case, the waiting request is a read to a container, which means that it can be woken up if entries are added to the same container. Since the event in our scenario says exactly that, the request is removed from the wait container (using the destroy operation) and written into the request container (6). From then follows the standard scenario, the request is again taken by the core processor (7), and this time it can be fulfilled, so the core processor generates a response including the read entries (8).

3.4.5.2 Types of Events

As for the types of possible events, the core model doesn't define which types of events exactly can occur. This is a thing to be decided by the implementation, as different implementations could go different ways in how detailed the information is that an event provides. Creating very detailed events can be more time consuming, but make it easier for the event processor to decide if a request should be woken up because of a certain event or not (reduce the amount of unneeded wake-ups). On the other hand, creating less detailed events will be easier but also increase the chance of waking up a request that can still not be processed. In that case, the request would just end up in the wait container again, but it would still use unnecessary processing time which would better be used for other operations.

The basic types of events that will probably in some way come to use in every implementation are:

- **Entries Added**
  When entries have been added to a certain container, operations must be woken up that want to do a read, take or destroy on that container and couldn't be fulfilled because there were not enough entries in the container.
- **Entries Removed**

When entries have been removed from a bounded container, operations must be woken up that want to write something into that container but couldn’t be fulfilled because there was not enough space left in the container for the entries to be written.

- **Lock Released**
  When a certain lock is released from a container or entry, operations must be woken up that want to access this container or entry but couldn’t be fulfilled because the container/entry was locked by another transaction.
- **Container Destroyed**
  When a container is destroyed, it is of course not possible any more to do any operations on this container. Because of that, requests that are waiting for anything concerning this container must be woken up. These operations will then respond to the user as failed because the container doesn’t exist anymore.

3.4.5.3 Timeout Handling

In the above case, the operation could be fulfilled before the request ran into a timeout. When this is not the case and a request runs into its timeout, it is handled by the timeout handler component. This is shown in Figure 24. Until the request is written into the wait container the scenario is the same as above. The timeout handler regularly (in a configurable time interval) checks the wait container if it contains any requests that have timed out (every request contains a timestamp for entering the core and a timeout value, so this can easily be checked) (3). Timed out requests are removed from the wait container (destroyed) and written into the request container (4), from where it is again taken by the core processor (5). The core processor checks every request if it has not timed out yet. If it has timed out, the core processor generates an error response (6) for that request that tells the user that the request could not be fulfilled within the given time, in other words the operation has failed.

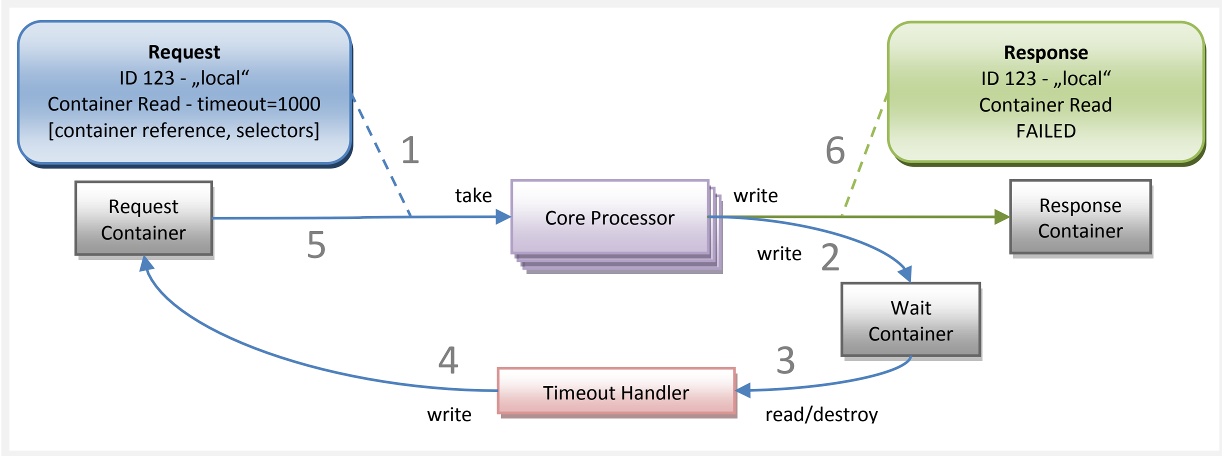


Figure 24: The core timeout handling structure.

To sum up the tasks of the event processing and timeout handling components: The event processor takes events from the event container. For an event it checks every request in the wait container if it is waiting for that event, removes such requests from the wait container and writes them back into the request container. The timeout handler regularly reads all waiting requests from the wait container and checks them for timeouts. It removes all timed out requests from the wait container and writes them into the request container. This structure does not only separate the blocking of requests from the core processor, it also grants extensibility: If in a certain usage scenario it would

happen very often that requests are delayed, the core could just run more than one event processor, because the processing of events can be done concurrently without problem.

3.4.6 Error Handling

Error handling is an important task when designing a system for which fault tolerance is a major requirement. When an error occurs during processing, the core processor must not crash, but create a response containing detailed information about the kind of error that occurred so the user is able to react properly to what happened. Defining already for the model which kinds of errors can occur in the core is especially important because every core implementation must later be able to understand error responses, no matter which core implementation they came from, so every implementation must keep to a single standard (otherwise interoperability would not be possible).

The following list defines all possible types of errors and when exactly they occur. Every error has a unique name that always begins with “Xvsm” and ends with “Exception” (since it is the most common standard to handle errors with exceptions):

- **XvsmContainerCreateException**
  Thrown when something goes wrong during container creation, e.g. the given size value is invalid or one of the container’s coordinators throws an error during initialization. In this case the parameters with which the containers were created should be checked for correctness.
- **XvsmContainerNotFoundException**
  This exception is thrown when the container to be used in an operation doesn’t exist (e.g. when it was deleted). This exception can be thrown by any operation that uses a container.
- **XvsmContainerWriteException**
  Thrown when something is wrong in a write operation (*write* or *shift*) concerning the given list of entries. This is the case when an entry contains selectors that don’t fit with the coordination types supported by the container, when an entry is written to a key in the meta container that is read only, or when the information contained in one of the selectors is not correct (e.g. when the index where an entry should be added in a *vector* coordinated container is invalid; if the information contained in a selector is correct is decided by the coordinator).
- **XvsmContainerReadException**
  Thrown when something is wrong in a read operation (*read*, *take* or *destroy*) concerning the given list of selectors. This is the case when a selector is used for reading that doesn’t match to any of the container’s coordination types (e.g. a selector for *fifo* on a container that supports only *key* coordination), or when the information contained in one of the selectors is not correct (if the information contained in a selector is correct is decided by the coordinator).
- **XvsmOperationTimeoutException**
  Thrown when the operation ran into a timeout while waiting for a lock, which is the case when wanting to write into a container that is currently locked by another transaction. This can happen in every operation that allows defining a timeout.
- **XvsmOperationFailedException**
  Thrown when the operation failed because it was unable to be fulfilled (although a lock could be acquired) and ran into a timeout. This can happen when there are not enough entries in

the container (in case of a read operation), or when there is not enough space left in the container (in case of a write operation), or when one of the container's coordinators doesn't allow an entry to be written (e.g. in a *key* coordinated container, when an entry should be written with a key that already exists).
Note: This is exception is similar to the one above, the important difference is that in the one above the operation didn't even come to finding out if the operation could be fulfilled when there wasn't a lock.

- **XvsmTransactionNotFoundException**
  Thrown when the transaction used by an operation doesn't exist (e.g. when it was already committed or rolled back). This can happen in every operation that uses a transaction, and of course at commit and rollback.
- **XvsmAspectException**
  Thrown in case of an aspect-related error. This is always the case when an aspect throws an exception and thereby gets the operation to fail. This error can be thrown in every of the core's operations since aspects can be added to every operation in the core.
- **XvsmCommunicationException**
  Thrown when something goes wrong while trying to communicate with another core, e.g. the other core cannot be reached. This error can be thrown in every operation that can communicate with a remote core (all operations concerning containers, transactions and/or aspects at remote cores).
- **XvsmSerializationException**
  Thrown when there is an error serializing or deserializing the data in a request or response (e.g. selectors or entries). This is mostly related to remote communication, but can also happen when communicating with the embedded core since entries could be stored in a container in a serialized state. Since all operations can also carry user-defined metadata for aspects with them (see chapter 5 for more information on that), this data can also be a reason for this exception to be thrown during an operation.
- **XvsmFatalCoreException**
  When this exception is thrown, something completely unexpected happened within the core. After this error is thrown the integrity of the core is no more guaranteed, it is especially likely that the part concerned by the operation (e.g. a certain container) is no more reliable.

For a core implementation in a programming language that supports inheritance, this feature should of course also be taken into account when implementing the given exceptions. All exceptions should have a single *XvsmException* base class. Also some of the exceptions are well suited to have a common base class, e.g. all the container related exceptions could be inherited from *XvsmContainerException*.

3.5 Architecture Variants

For different usage situations of the core, there are several possible architecture variants.

3.5.1 Peer

The standard architecture variant is the peer. It contains all of the core's earlier introduced functionality. A core is running underneath an application (normally started by the application). The application accesses the core by use of an API lying over the core. This could be the core's low level

API, but could also be a higher level API on top of the core to provide easier use of the core's functions (especially when the core is extended by profiles, see chapter 3.6). Underneath the core lies the communication layer that defines the communication mechanisms that are used by the core to communicate with other cores. This is nothing else than the sender and receiver components in the core structure (as it has been said, these components are built to be replaceable to support different communication mechanisms). All communication done by the application with other peers goes through the core.

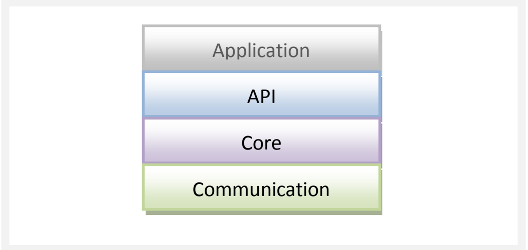


Figure 25: Peer architecture.

3.5.2 Client

The client architecture is very similar to the peer. The difference is that it does not have a fully functional core, only a "client core". This client core is more or less only there to fill the gap between the API and the communication layer, it cannot manage any data locally. According to the core structure that has been introduced earlier, this variant of the core could be seen as only consisting of the embedded API, the request and response containers and the sender and receiver components (because it has no core processor and event processing components it cannot process any requests locally). This variant is useful when the machine the application runs on has very little power and/or a connection with very low bandwidth (e.g. a mobile device), and because of that should not/cannot host any data itself.

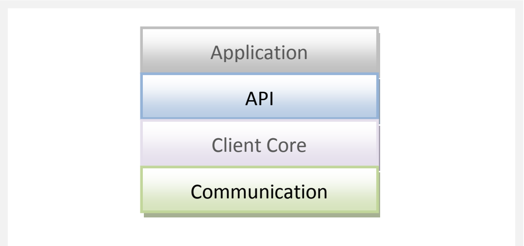


Figure 26: Client architecture.

3.5.3 Standalone

The core can not only serve for communicating with other cores, but also for locally coordinating the data structures of an application, in which case it is not necessary to have any remote communication functionality. The standalone variant serves that purpose. Without a communication layer, the core cannot communicate with other cores and vice versa, and can only be accessed directly from the overlying application. According to the core structure, this would be a core without the sender and receiver components, which means that it cannot receive and process any remote requests or responses.

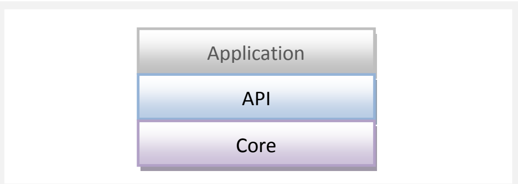


Figure 27: Standalone architecture.

3.6 Profiles

One of the goals of the core is to provide a very compact set of only the most important functions, but also be highly extensible, so that new functions can easily be added with modules extending the core.

These modules are called *profiles*. Profiles are situated directly above the core layer, underneath the API layer, with the API being specialized on using the core together with the configured profiles. In this way, the user of the core sees the core as a single system, although it may be extended by multiple profiles.

As shown in Figure 28, there are many possibilities for useful profiles. For example, a core specialized on security would probably need authentication and authorization mechanisms as well as encrypted communication between cores, for which it could be extended by appropriate security profiles.

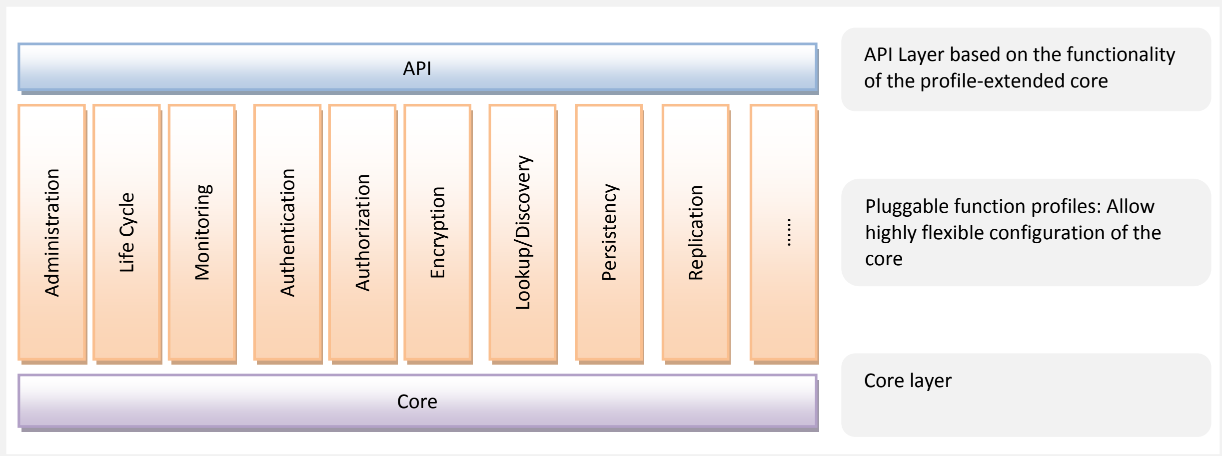


Figure 28: Extending the core with profiles.

The main way of adding these profiles to the core is by the use of aspects. Aspects are pieces of code that can be added into the core to be executed directly before or after any of the core's operations. So the profiles are actually not only added on top of the core but (at least partly) within the core itself. An aspect for logging could for example simply be one that is added before the execution of any operation (every *pre ipoint*). This leads to calling the aspect every time before an operation is executed, and the aspect could log any information available about that operation. This is of course a very simple example, many of the profiles listed above would probably need several different types of aspects to implement their functionality into the core.

A more complex example would be an authentication profile. It is one that shows very well that a profile itself can efficiently use the space to manage its own data. In this case, the profile could use containers in the local space to manage lists of users and user groups. The owner, as well as access permissions, could be stored for every container in its meta container. Aspects on pre ipoints could

implement user authentication mechanisms that prevent users from accessing containers that they don’t have the right to. Even further, aspects could control access to all of the core’s functions, like creation of containers and transactions. For best usage of these additional functions, the API layer would have to provide appropriate functions for authentication, creating users, and so on.

An implementation of a simple security aspect for the .Net version of the core is introduced in chapter 5. For a detailed description of aspects in the XVSM core, see [13].

4 The XcoSpaces .Net Kernel Implementation

4.1 Introduction

This chapter gives a detailed explanation about the implementation of the XVSM core using the C# programming language of the .Net Framework [34]. The name of this XVSM .Net implementation is *XcoSpaces*. The phrase "Xco" stands for *Xcoordination*, a term that addresses the space's ability of extensible coordination and is also the name of the company [35] that is later planned to distribute XcoSpaces. For the core the XcoSpaces implementation uses the term *kernel*, because the XcoSpaces developers found it to be more fitting when looking at the core as a piece of software. Another advantage to the renaming is that is it's clear that when speaking of the *core* the theoretical model is meant and when speaking of the *kernel* it means the XcoSpaces implementation.

After *MozartSpaces* [36], the open source XVSM implementation developed in Java (which is also described in [6] and [13]), XcoSpaces is the second implementation of XVSM. Because XcoSpaces primarily targets the Microsoft community and the Windows operating system, .Net C# has been chosen as the used programming language. Further reasons for C# are that it is a very modern and high-level programming language and supports the implementation of the XVSM concept very well. Also C# is one of the most wide spread programming languages and is used by a large number of developers, which is an ideal prerequisite for spreading the XVSM paradigm. Visual Studio 2005 has been chosen as programming environment, because it is most common for C#, supports all of C#'s features, is widely available and supported very well.

To name a few facts, the XcoSpaces kernel consists of about 160 classes, spread over 11 assemblies (single projects). All classes together have about 10000 lines of code and an additional 6000 lines of documentation. The kernel's testing environment currently consists of 564 unit tests which give a near to complete code coverage (about 95%).

In addition to the kernel, the *high level API*, which is developed by Ralf Westphal, is the second part of the XcoSpaces implementation. This API wraps the kernel and provides more convenient classes and methods (like classes that implement *fifo* or *key* coordinated containers, *XcoQueue* and *XcoDictionary*), but doesn't support all of the kernel's features. Since this thesis only concentrates on the structure and implementation of the kernel, the high level API will not be discussed further.

The following chapters show how the conceptual XVSM core model has been implemented, which changes there are in comparison to the model and which things have been uniquely added to the XcoSpaces kernel that were not part of the core model, as well as implementation specific details like class and interface descriptions and the project structure.

4.2 The Kernel Architecture

The first thing to take a closer look at is the architecture of the kernel. Compared to the core model, the kernel structure has undergone some changes. A bit of the clearness and encapsulation of the model has been given up to be able to provide the highest possible level of performance.

The kernel structure can be seen in Figure 29. It shows the components named by the names of the classes that implement them (or at least by the name of the representing class, if a component

consists of more than one class). Because of being the class that represents the whole kernel, the embedded API has been named *XcoKernel* (described in chapter 4.2.1). The sender and receiver components have been pulled together into a single component *CommunicationService* (see 4.2.2). This component is easily replaceable for using different communication service implementations. The request and response containers have been removed because it turned out to be really difficult to get them performing well, and removing them improved the processing speed of local requests significantly. Because the job of the request and response containers is also the automatic distribution of local and remote messages to different components, this task now also needs to be done by other components: The XcoKernel communicates directly with the CommunicationService for remote requests and with the CoreProcessor (see chapter 4.2.5) for local requests (the requests are added directly into the core processor's thread pool queue). The CommunicationService gives requests also directly to the CoreProcessor and responses to the XcoKernel. Instead of the response container a *ResponseDistributor* component has been added which simply passes remote responses to the CommunicationService and local responses to the XcoKernel.

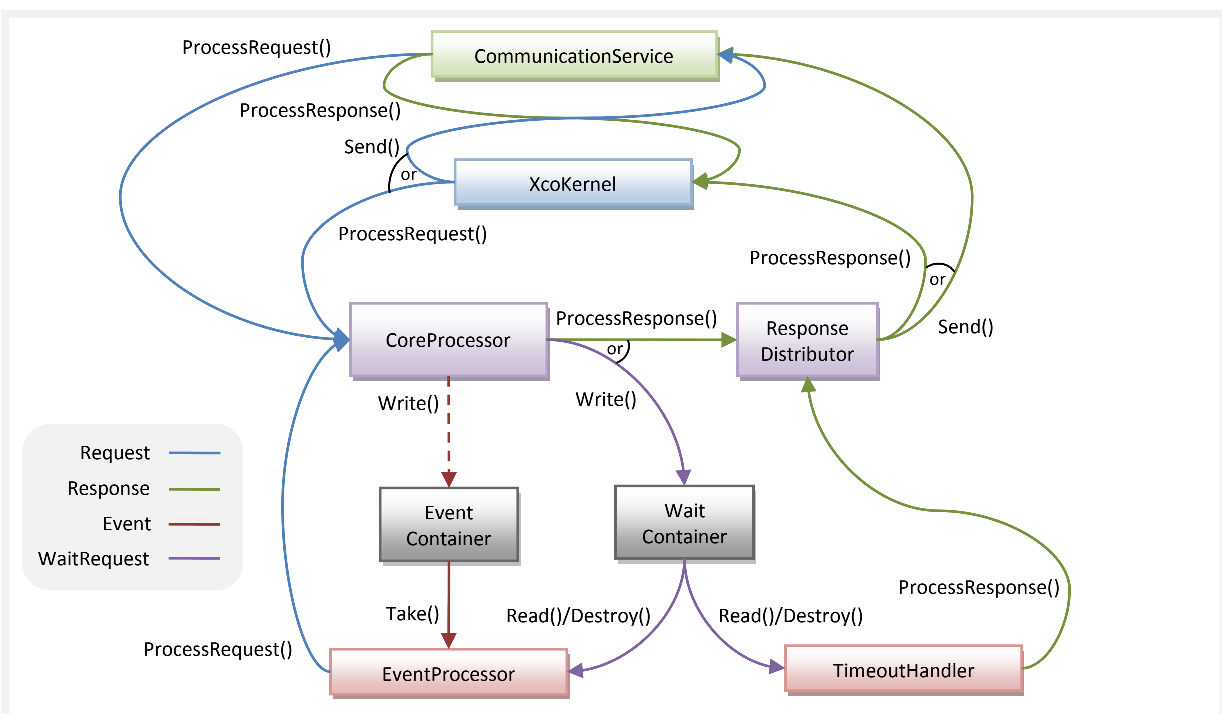


Figure 29: The XcoSpaces kernel structure.

Small changes have also been made to the event processing components: When delaying a request, the CoreProcessor not only writes the request into the wait container, but wraps the request into a *WaitRequest* message (see chapter 4.2.4.3) that additionally contains information about what this request is waiting for, so that it easier to decide for the EventProcessor (see chapter 4.2.6) if a certain request should be woken up. Another difference is that the TimeoutHandler (also handled in chapter 4.2.6) directly creates responses and gives them to the ResponseDistributor instead of letting the CoreProcessor handle that (which also adds a bit of performance since the request doesn't need to go all the way through the core again).

After this short overview, the following chapters provide more detailed information about the implementation, functionality and structure of the main kernel components.

4.2.1 The Embedded API: XcoKernel

The embedded API is implemented in a single class named *XcoKernel*. From an outside view the XcoKernel class represents the kernel itself (or in fact IS the kernel), since the space is not only accessed by using the XcoKernel, but is also created by instantiating the XcoKernel. The API provided by the XcoKernel provides methods for synchronous calls to every kernel function. It contains no asynchronous methods of any kind. It has been paid attention to couple the XcoKernel class as loosely as possible with the rest of the core, so if an asynchronous API would be needed the XcoKernel class could be replaced without much effort, or simply be extended by inheriting it. Next to the XcoKernel, there are some other important classes that are needed to work with the kernel (because they are needed by the XcoKernel's methods) and therefore can be seen as part of the API:

ContainerReference

The reference to a container that is generated when the container is created and can be used to access this container. It is needed by many of the XcoKernel's methods like Read(…), Take(…), Destroy(…), Write(…) and Shift(…).

Selector

The base class for selectors that are used to specify which entries are read in a read operation, or to specify the coordination properties when writing entries to a container (like a key value or vector index). For every available coordination type there is a special class belonging to that coordination type that inherits the Selector class, like the *FifoSelector* for *fifo* coordination.

Entry

Entries can be stored in containers, so this class is used every time something has to be written into or read from a container. An entry holds the stored value (object) and a list of selectors that define how this entry is coordinated in the container.

TransactionReference

The reference to a transaction that is generated when the transaction is created and is needed for every operation that should be done within this transaction, as well as to commit or roll back the transaction.

Notification

The class that represents a notification (a short introduction to notifications as a core requirement was given in chapter 3.2.2.2). This class allows to be automatically notified when something changes in a container. The implementation of notifications in the XcoSpaces kernel is handled later in detail.

The use of the API is best shown by a small example. It shows creating a *fifo* coordinated container, writing an entry into this container, taking the entry and then destroying the container (because the kernel is instantiated with the *using* statement, it is automatically disposed at the end):

```
//create a new space by instantiating the kernel
using (XcoKernel kernel = new XcoKernel())
{
	//create a new fifo coordinated container
	ContainerReference cref = kernel.CreateContainer(
		null, -1, false, new FifoSelector());

	//write an entry into the container
	kernel.Write(cref, null, 1000, new Entry("Hello Space!"));

	//now read the entry from the container and write the value to the console
	List<IEntry> result = kernel.Take(cref, null, 1000, new FifoSelector(1));
```

```
        Console.WriteLine(result[0].Value.ToString());

        //destroy the container
        kernel.DestroyContainer(cref);
}
```
Code Example 1: Working with the XcoKernel.

A more detailed description of the XcoSpaces kernel API, its methods and how they are used can be found in[12].

4.2.2 Remote Communication

The remote communication components have changed in comparison to the core model by merging the sender and receiver into one single component, the *communication service*. This has been done to present the communication service as a single point of replacement when wanting to replace the default communication service by another one. The main requirements for this component have not changed, one being that the communication service must be replaceable and the other being that the kernel must provide a default communication service that is fitting to as many usage situations as possible (being optimized for both reliability and performance).

For implementing this default communication service it was chosen to use *WCF* (the *Windows Communication Foundation* [37]) which is a part of .Net Framework 3.0. The reasons for this choice are that WCF already provides communication functions on a very high level and is therefore very easy to use and implement, and that WCF itself is highly configurable which allows it to be fitting for many usage situations. It can for example be configured to communicate over TCP, but it could as well use web services and SOAP for communication, just by changing a few configuration parameters. It has therefore been assured when implementing the WCF communication service for the kernel that none of WCF's configurability gets lost.

The replaceability of the communication service is ensured by the use of contracts. More details on contracts in the kernel and the communication service contract are given in chapter 4.4. More detailed information about communication services in the kernel and the WCF implementation can be found[12].

4.2.3 Core Containers

The core container is implemented as a generic class and thus perfectly suitable for both the event and the wait container (using Event/WaitRequest as generic type). Genericity is also very useful here because of the type safety that is reached with it for the messages that are written into and read from a core container. Every component that is using the core container can rely on that it only contains objects of that certain type. Like a normal container, the core container also uses entries to store its values. But because the normal entry has no type safety (its value is of type *object*) a special generic subclass is used:

Entry<T> : Entry
Special generic variant of the entry class that is a subclass of *Entry*. The values stored in this entry can only be of the type T. This class is used for core containers only.

Since the request and response containers have been removed in the kernel structure, it is no more needed for the core container to support the kind of labeling entries (with "local" or "remote") as it is needed in the core model. Thus the core container's functionality is very close to a blocking

message queue, with the additional abilities to read all entries and destroy a certain entry. Despite of that, the methods that are provided for writing to and reading from the core container very much stick to the definitions of the core model:

CoreContainer<T>	
WriteOne	Writes one Entry<T> into the container.
TakeOne	Takes one Entry<T> from the container. If there are no entries in the container, the method blocks until there are entries in the container to be taken.
ReadAll	Reads all Entries from the container and returns a List<Entry<T>>. An empty list is returned if there are no entries in the container.
DestroyOne	Removes a certain Entry<T> from the container. If the entry was really removed *true* is returned, else *false* (this is needed for concurrency issues between event processor and timeout handler).

It is also important for the core container to coordinate concurrent access from different components. The core container must not crash if for example a *WriteOne* and a *TakeOne* operation are executed by different threads at the same time. To solve this problem the core container makes use of the *System.Threading.ReaderWriterLock* class which is able to coordinate multiple threads using the multiple-reader/single-writer principle and is therefore perfectly suitable for the given situation. While *ReadAll* must acquire a reader lock, *WriteOne*, *TakeOne* and *DestroyOne* must acquire a writer lock when wanting to access the container. Figure 30 shows a class diagram of the core container structure:

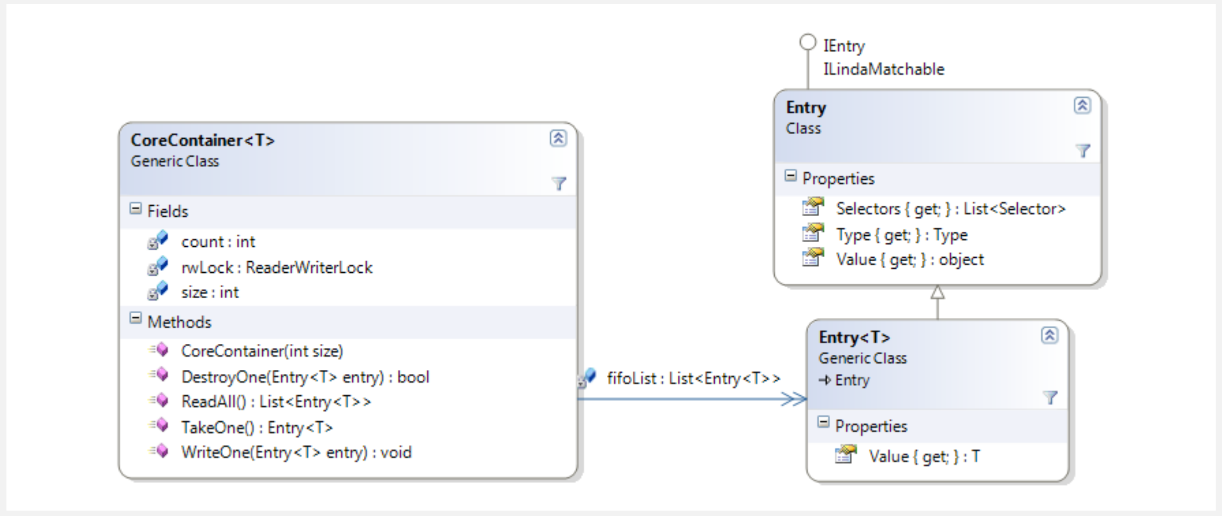


Figure 30: Class diagram of the kernel's core container.

4.2.4 Message Types

There are four different types of messages in the kernel that are communicated between the kernel's components: *Request*, *Response*, *Event* and *WaitRequest* (which is an addition to the original core model). In the kernel these messages are implemented as different classes.

4.2.4.1 Request

The following class diagram shows an overview of both request and response messages:

Figure 31: Class diagram of the kernel's request/response message classes.

The request is implemented in the *RequestMessage* class. It inherits from the abstract class *Message* which contains parameters that are equal for both request and response messages.

Message	
MessageID	The id of the message assigned and used by the XcoKernel to find the correct Response to a Request sent to the kernel (the response from the kernel comes back with the same message id).
IsRemote	True if the address of the message belongs to a remote space.
RemoteAddress	The address of the remote space where this message should be sent to, or has been received from. Or null, if it is the local space.

RequestMessage : Message	
Content	The content of this request message in form of a *RequestMessageContent* object.
Timestamp	The date when the request entered the kernel. This is important for checking if a request has reached its timeout.

The RequestMessage class doesn't directly contain the information that is needed for the operation to be executed in the kernel. This information is contained in a *RequestMessageContent* object. The reason why the content is separated from the message itself in that way is that this is needed for fault tolerant remote communication. When sending a request to another kernel, the RequestMessage object must be serialized. If the request contained data that causes deserialization on the other side to fail, the remote kernel would be unable to even send back an appropriate error message, because it cannot get the message id and remote address contained in the message. By separating the content from the message this can be avoided: The content is serialized first and stored in the request as byte array. Since the message itself doesn't contain any other data that could cause the deserialization to fail, the problem of a message that cannot be deserialized cannot occur any more. If after deserializing the request, the content cannot be deserialized, it is no problem for kernel any more to generate a response with a corresponding error.

The *RequestMessageContent* class itself doesn't have any parameters but serves as base class for all different contents a request can have. For every operation (like container create, container destroy, read, write, and so on) there is a special subclass of *RequestMessageContent* that contains all the information that is needed for that certain operation. Although this solution requires a lot of class definitions, it has been chosen because it allows a clear definition which parameters are needed by every different operation. It is not necessary to give a detailed explanation of all these classes here, but just to show an examples, this is how the content class for the *container create* operation looks like:

RequestContainerCreate : RequestMessageContent	
Size	The size of the container.
Unique	True if entries in the container should be unique.
CoordinationTypes	A List<Selector> that defines the coordination types for the container.

4.2.4.2 Response

Since the response also transports data between the XcoKernel and CoreProcessor components, it is to some degree similar to the request. This is represented through the fact that like the RequestMessage, the *ResponseMessage* class also inherits from the *Message* class (thereby inheriting its *MessagID*, *IsRemote* and *RemoteAddress* properties), as it is also displayed in Figure 31.

ResponseMessage : Message	
Content	The content of this response message in form of a *ResponseMessageContent* object.

Also similar to the request (and of the same reasons), the response message and the content are separated. Unlike the request, the response doesn't need a timestamp since timeouts are not a topic for responses. Since all response contents share the fact that they must be able to return an error, a property for doing that is present in the *ResponseMessageContent* class:

ResponseMessageContent	
Error	Exception that has been thrown in the kernel if an error occurred while processing the corresponding request, or null if no error occurred.

Directly sending back an exception (meaning an instance of the *Exception* class) in the response when an error occurred provides a very comfortable way for the XcoKernel to give the error back to the overlying application. If the *Error* property of the response is not null, the XcoKernel just has to rethrow the exception in the thread where the XcoKernel's method was called.

Just like with the RequestMessageContent, for every different type of operation there exists one class inheriting the ResponseMessageContent, defining which parameters the response for that operation contains. Corresponding to the example given above, the response content class for the *container create* operation looks like that:

ResponseContainerCreate : ResponseMessageContent	
CRef	The *ContainerReference* object that can be used to access the container that has been created.

4.2.4.3 WaitRequest

Since a WaitRequest only stays within the kernel and is never the target of serialization or deserialization it doesn't need most of the properties that were needed for request and response. The reason for not writing a request directly into the wait container is that by wrapping it into a WaitRequest it can be equipped with additional information that makes determining if a request should be woken up much easier.

WaitRequest	
Operation	The waiting request.
CRef	Reference to the container that is the target of the operation.
TRef	Reference to the transaction in which the operation was executed (or null if the transaction has no importance for the reason of waiting).
Type	The type of event that the request is waiting for (like *EntryAdd* or *EntryRemove*).

A waiting request is normally woken up when the type of event that occurred matches the type of event that the request is waiting for and the container for which the event occurred is also the same as the one that is the target of the waiting operation. The transaction only plays a role in special cases.

4.2.4.4 Event

An event must contain all information that is needed to determine if a waiting request should be woken up. Because of that the event must actually be able to store the same information as the WaitRequest. But because the information contained in the event is of clearly different purpose, it was decided to implement two different classes either way. An event is represented by the *WaitEvent* class:

WaitEvent	
Operation	The operation that should be woken up, if the event's target is to wake up only one certain operation (otherwise null).
CRef	The reference to the container for which the event occurred.
TRef	The reference to the transaction for which the event occurred (or null if the transaction has no importance for the event).
Type	The type of event that occurred (like *EntryAdd* or *EntryRemove*).

The following figure presents the class structure of the wait/event classes (The *RequestContainerOp* class is the base class for all request messages that are allowed to go into waiting state, namely all requests that concern operations on a container):

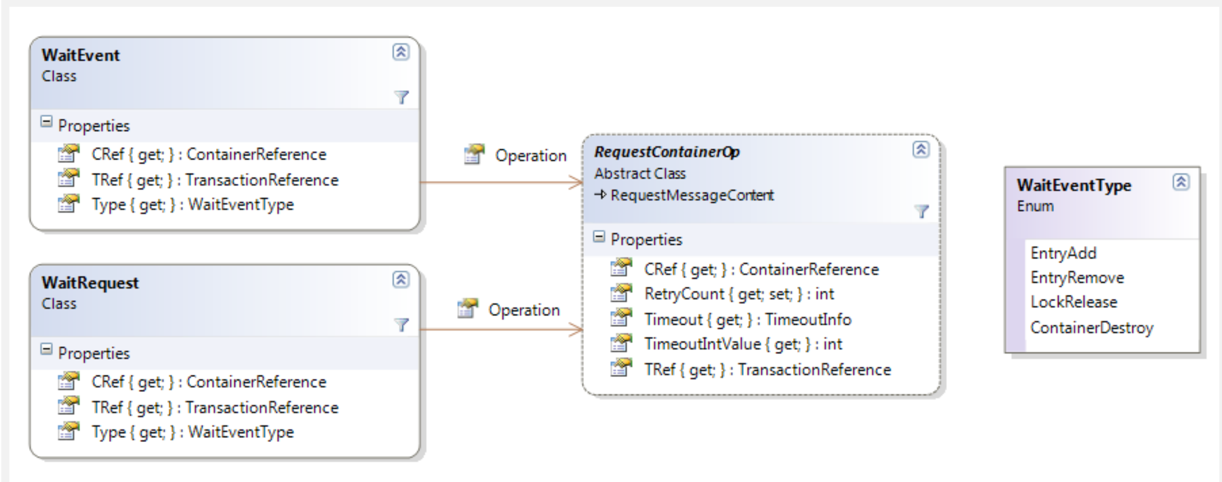


Figure 32: Class diagram of the kernel's wait/event message classes.

The details about waiting and event processing are handled in chapter 4.2.6.

4.2.5 The CoreProcessor

With the removal of both the request and the response container, the structure of the core processor has changed to some degree. Since the requests are now given directly to the core processor by the other components (the XcoKernel and the communication service, as displayed in Figure 29), the core processor itself has to deal with dispatching the requests to certain processor threads. For this task a component is added to the core processor called *thread dispatcher*. Its task is simply giving an incoming request to a processor thread that is currently free, or queue the requests as long as all processor threads are occupied.

When the kernel, it became clear that there were two good solutions for implementing the thread dispatcher, and both of them had their advantages. The first was implementing the thread dispatcher by using the *System.Threading.ThreadPool* class, with the advantage of having a solution that doesn't need any additional libraries and is directly a part of and supported by the .Net Framework, but the disadvantage that the thread pool provides nearly no possibilities for specific configuration. The second solution was using the *Concurrency and Coordination Runtime (CCR)* [38] that is a specialized library for executing concurrent operations and thread coordination and is distributed together with Microsoft's *Robotics Framework*. The advantage to this solution is clearly that it is specialized at exactly the task that it is needed for in the kernel and that it is also very well configurable, but it is also an additional library that is underlying certain license agreements that have to be taken into consideration.

Because of that it has been decided to make the thread dispatcher a replaceable component (like the communication service). The second reason is that the thread dispatcher is one of the main points in the kernel to ensure scalability. Intelligent and fast processing of requests on both low-end and high-end systems is crucial for the kernel's performance. By having the thread dispatcher replaceable users could even develop specialized thread dispatchers for certain hardware or operating systems.

The replaceability of the thread dispatcher is ensured by the use of contracts. More details on contracts in the kernel and the thread dispatcher contract are given in chapter 4.4.

Figure 33 shows the most important classes of the core processor in a class diagram:

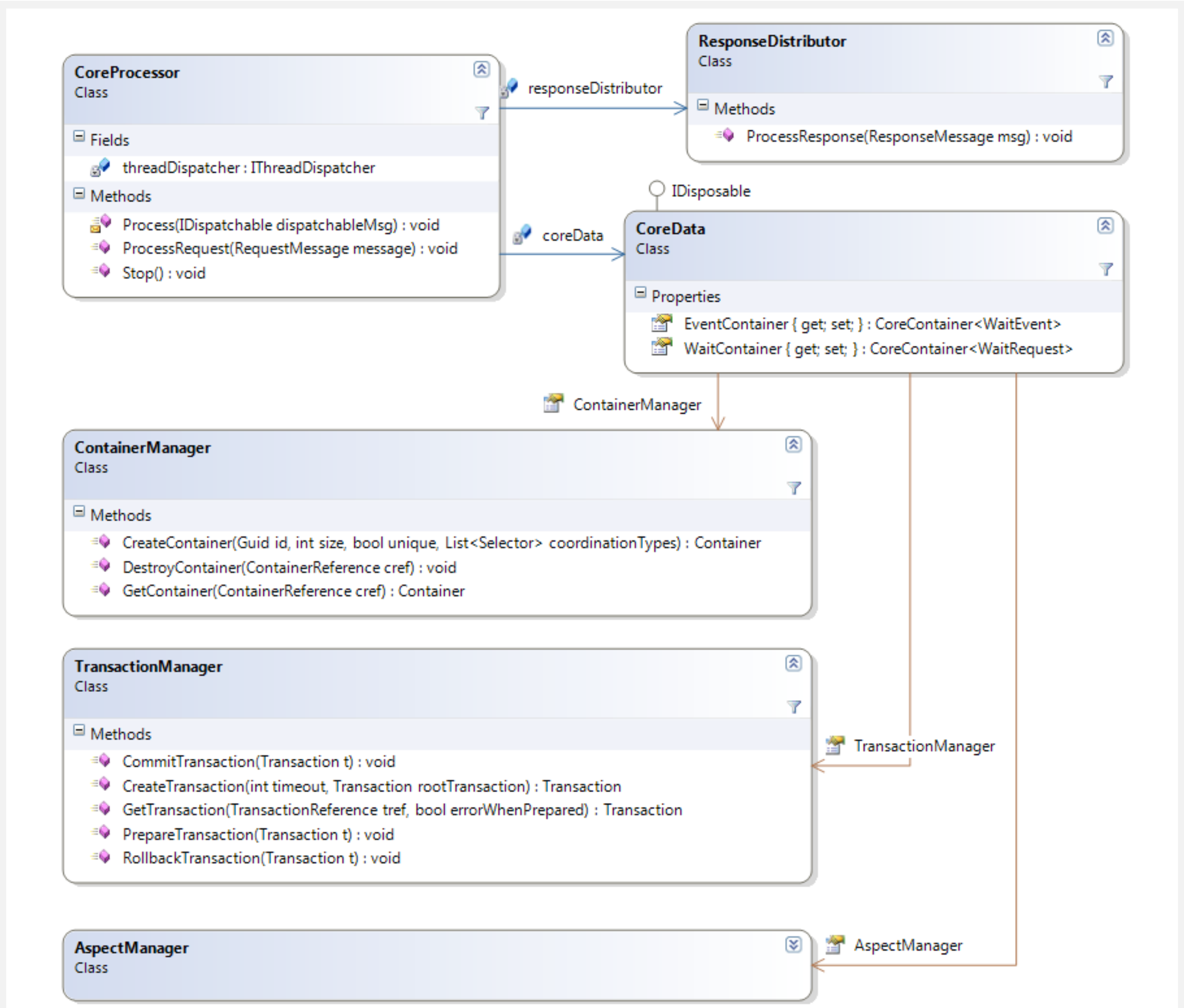


Figure 33: Class diagram of the kernel's core processor classes.

Figure 34 shows the structure of the core processor. Instead of the processors taking requests from the request containers, they are now actively called by the thread dispatcher. Instead of writing the response into the response container, the processor threads now call the response distributor. This includes another very important difference that is not visible in the figure: The response distributor is a component that doesn't run in an own thread. When being called, it does all its processing directly in the thread of the processor that called it. This also means that the processing of the response is all actually done in the processor thread until it either reaches the XcoKernel or is sent by the communication service. Although this makes the overall structure less clean, this is a main reason that the processing is significantly faster than with a response container, because it removes any indirections on the way of the response from the processor to its destination (the coordination of threads is unfortunately very costly).

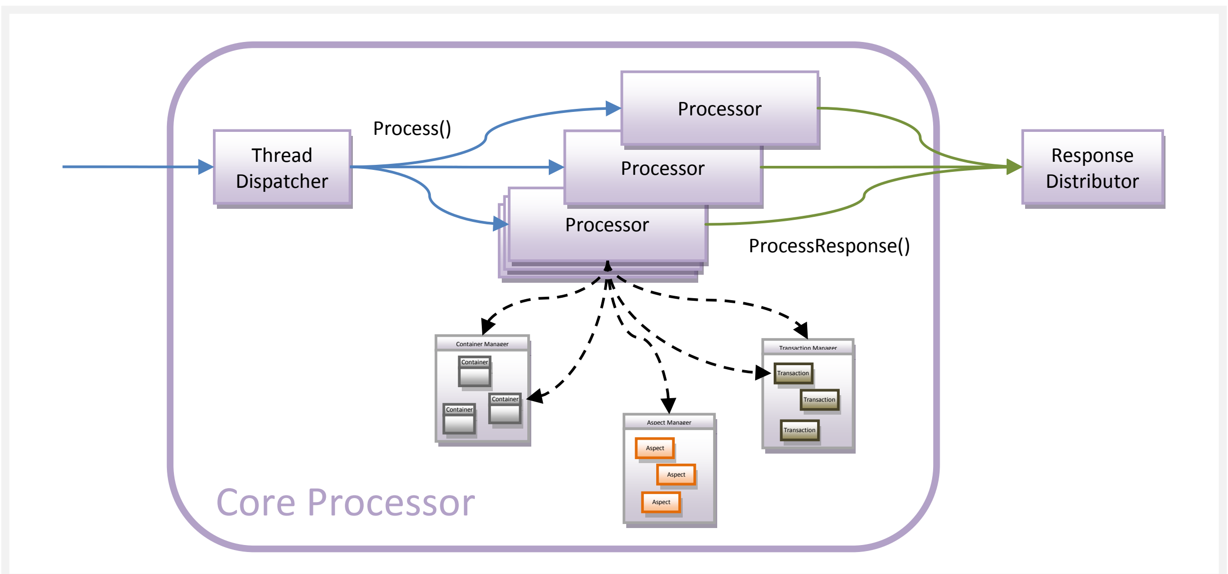


Figure 34: The structure of the kernel's core processor.

Despite of that the core processor structure is mainly the same as in the core model (which is why the other parts are only shown very small in the above figure). The *container manager* and *transaction manager* provide thread safe methods for accessing containers and transactions:

ContainerManager	
CreateContainer	Creates a new container according to the given size, uniqueness and list of coordination types. A new ContainerReference is created for the container, and the container is added to the local space. In addition a meta container (a *key* coordinated container) is created for this container.
DestroyContainer	Removes the container that belongs to the given ContainerReference from the local space. An exception is thrown if a container with this reference doesn't exist.
GetContainer	Gets the *Container* object that the given ContainerReference belongs to. An exception is thrown if a container with this reference doesn't exist.

TransactionManager	
CreateTransaction	Creates a new transaction. A new TransactionReference is created for the transaction, and the transaction is added to the local space.
CommitTransaction	Commits the transaction by committing all operations contained in the transaction's log and releasing all locks that are owned by this transaction, and removes it from the local space.
RollbackTransaction	Rolls back the transaction by undoing all operations contained in the transaction's log and releasing all locks that are owned by this transaction, and removes it from the local space.
GetTransaction	Gets the *Transaction* object that the given TransactionReference belongs to. An exception is thrown if a transaction with this reference doesn't exist.

The aspect manager has undergone some changes compared to the core model. These changes are handled in detail in chapter 5 which deals with the implementation of aspects in the XcoSpaces kernel. A detailed explanation of the implementation of containers and transactions (meaning the *Container* and *Transaction* classes) is given in chapters 4.2.7 and 4.2.8.

Finally, the *CoreProcessor* class itself only needs to show a single method to the outside:

CoreProcessor	
ProcessRequest	Takes a RequestMessage and gives it to the thread dispatcher for processing.

4.2.6 Event Processing and Timeout Handling

Despite of wrapping the waiting requests by the *WaitRequest* class and letting the timeout manager create responses directly, the event processing and timeout handling components haven't undergone many changes compared to the core model. The types of waiting (*EntryAdd*, *EntryRemove*, *LockRelease* and *ContainerDestroy*, see chapter 3.4.5.2) remain the same. Most of the effort when implementing this part of the kernel has gone into improving the performance of generating events and delaying and waking up requests as much as possible (the part of the model that doesn't provide a detailed definition). This was tried to be reached by also optimizing the core processor so that events are only generated when they are necessary (but still assuring that these optimizations don't noticeably slow down the normal processing of the kernel).

The main point of optimization is the container: Since all waiting requests are waiting for some event that is connected to a certain container, there is no need to generate events if there are no requests waiting for that container. Also, if entries are added to the container within a certain transaction, only requests that are waiting for *EntryAdd* on the same container AND using the same transaction can be woken up immediately. Operations that are not using the same transaction can only be woken up as soon as the transaction is committed, because until that they will not be able to access anything that has been added to the container (because everything that has been added is still locked by another transaction) (the same goes for *EntryRemove*). This problem is solved through the use of the *WaitCounter* class. It is used to count how many requests are waiting for *EntryAdd* or *EntryRemove* on a certain container, meaning every container has two *WaitCounter* objects, one for each of these event types (also see Figure 35 for an illustration of the WaitCounter in a class diagram).

WaitCounter	
Increase	Increases the wait counter by 1 for the given TransactionReference.
Decrease	Decreases the wait counter by 1 for the given TransactionReference.
Is0	If called with a TransactionReference, returns true if the wait counter for this transaction is 0. If called with no parameters, returns true if the overall wait counter is 0 (means no waiting requests for any transaction).

Every time a request is delayed, the *Increase* method of the appropriate WaitCounter is called. Every time the event processor or timeout handler removes a waiting request, it calls the *Decrease* method of the same WaitCounter. Whenever entries are added to or removed from a container, the core processor checks by calling *Is0* for the current transaction if an event needs to be created (the generated event then contains the TransactionReference of the current transaction). Whenever a transaction is committed in which entries are added to or removed from a container, the core processor calls *Is0* without parameters from the appropriate WaitCounter of the affected container to check if an event has to be generated (the generated event then contains no TransactionReference), because after committing the transaction no lock is being held on the container so that all waiting operations regardless of which transaction they use can be woken up.

The handling of the *EntryAdd* and *EntryRemove* events in the event processor is simple: If one of these types of events is received, all waiting requests are woken up that wait for the same type of event and the same ContainerReference, and, if the event contains a TransactionReference, that have the same TransactionReference (otherwise this value doesn't matter).

The handling of the *ContainerDestroy* event is even simpler: The event is generated every time a container is destroyed and one of the container's WaitCounters returns false when calling *Is0* without paremeters (meaning there is at least one waiting request). When the event processor receives such an event, it wakes up all waiting requests with this ContainerReference.

More complicated is the waking up of requests that are waiting for the release of a lock, especially when locking is done at entry level (so not the whole container is locked by a transaction, but only single entries). A very complex situation (that the kernel must nevertheless be able to handle) is for example the following: When a container is working with entry level locking (like one with *key* coordination, as said earlier the possible level of locking depends on the used coordination type) it is possible that different entries in this container are locked by read and write locks from different transactions. But if now an operation wants to read the entry count of the container, still the whole container (and not only single entries) has to be (read) locked, because as long as the transaction that read the entry count is not committed or rolled back, the container's entry count must not change. This also means that for being able to set a read lock on the container, there must not be any write locks on entries of this container. So, if such an operation is delayed, it has to be woken up as soon as all write locks on all entries of this container are released as long as they don't belong to the same transaction (because write locks of the same transaction of course don't affect the operation).

These kinds of complex situations make it very problematic to generate events at the right time and make them wake up the right requests. To still make the kernel reliable in such situations, the main part of handling the waiting for locks has been implemented directly into the kernel's locking mechanisms (centered in the *ContainerLevelLockManager* and *EntryLevelLockManager* classes). Although this reduces the kernel's performance a little bit, it has been decided that reliability is the more important thing here, since it would be fatal if a situation could be produced where a waiting request is not woken up correctly (and in the worst case, when running with infinite timeout, ends up being stuck in the wait container forever). The locking classes remember every request waiting for *LockRelease* and every time a lock is released check if the preconditions for waking up any of them are fulfilled. As soon as that is the case, a *LockRelease* event is generated containing exactly the request that should be woken up in the event's *Operation* property, which causes the event processor to wake up exactly this one event (as long as it is still in the wait container and hasn't timed out yet). A detailed description about the kernel's locking functionality can be found in [12].

4.2.7 Containers and Coordination Types

Within the kernel, containers are represented by the *Container* class, which is one of the kernel's most complex classes. It is not only responsible for all container operations that are executed on a certain container, but also for writing requests that cannot be fulfilled to the wait container, generating events concerning entries that have been added to and removed from this container if needed, managing the locking of the container and its entries (and also deciding at which level the locking for a certain operation has to be done), coordinating concurrent calls to the same container from different processors and managing the container's aspects.

4.2.7.1 The Container Class

The Container class provides methods for every container operation defined in the core model:

Container	
CRef	The ContainerReference object that is used to address the container from outside the kernel.
AddWaitCounter	The wait counter for requests waiting for *EntryAdd*.
RemoveWaitCounter	The wait counter for requests waiting for *EntryRemove*.
Read	Reads a list of entries from the container and acquires a read lock for the resources that need to be locked. Which entries are read is decided by the list of Selectors provided by the operation.
Take	Takes a list of entries from the container and acquires a write lock for the resources that need to be locked. Which entries are taken is decided by the list of Selectors provided by the operation.
Destroy	Removes a list of entries from the container and acquires a write lock for the resources that need to be locked. Which entries are removed is decided by the list of Selectors provided by the operation.
Write	Writes a list of Entries into the container and acquires a write lock for the resources that need to be locked.
Shift	Writes a list of Entries into the container and acquires a write lock for the resources that need to be locked. All entries that would prevent the operation from being fulfilled are removed from the container.
GetProperty	Reads the property with the given name from the container and acquires a read lock on the container if reading this property needs one. The type of the returned property value is specific based on which property is read. An exception is thrown if the name of the property is not known.

All these operations take a request (an object of a subclass of *RequestMessageContent* that belongs especially to that operation, e.g. a *RequestRead* object in case of a read operation) and a transaction as input parameters. Note that all container operations within the kernel run with transactions, even if the operation was executed from outside the kernel without one. For every container operation that enters the kernel without a transaction, an *implicit transaction* is created and used for only that operation (meaning the transaction is created directly before the operation's execution and committed directly after if the operation was successful, or rolled back if the operation has failed).

One method that was originally not intended by any function in the core model is the *GetProperty* method. It provides access to properties of the container like its maximum size and the current entry count. While being able to read a container's maximum size can be seen as a useful addition, reading the entry count is a thing that is absolutely necessary for using a container in many situations. The theoretical model intends to use the *meta container* for that. But with the kernel implementation, this brings some problems: First, data in the meta container cannot be made write protected to the outside (or at least not in an easy way), so the count could be overwritten from the outside. Second, although writing the count to the meta container every time entries are added or removed would be not a big problem, it would at least slow down the container operations.

Therefore, to prevent the expense of maintaining these properties in the meta container, the GetProperty method has been implemented to be able read the properties directly from the container (properties that can be read are the maximum size, the entry count and the list of

coordination types). The method not only has its use for the container itself, but also for the container's coordination types, more on that later.

Figure 35 shows the container structure in a class diagram:

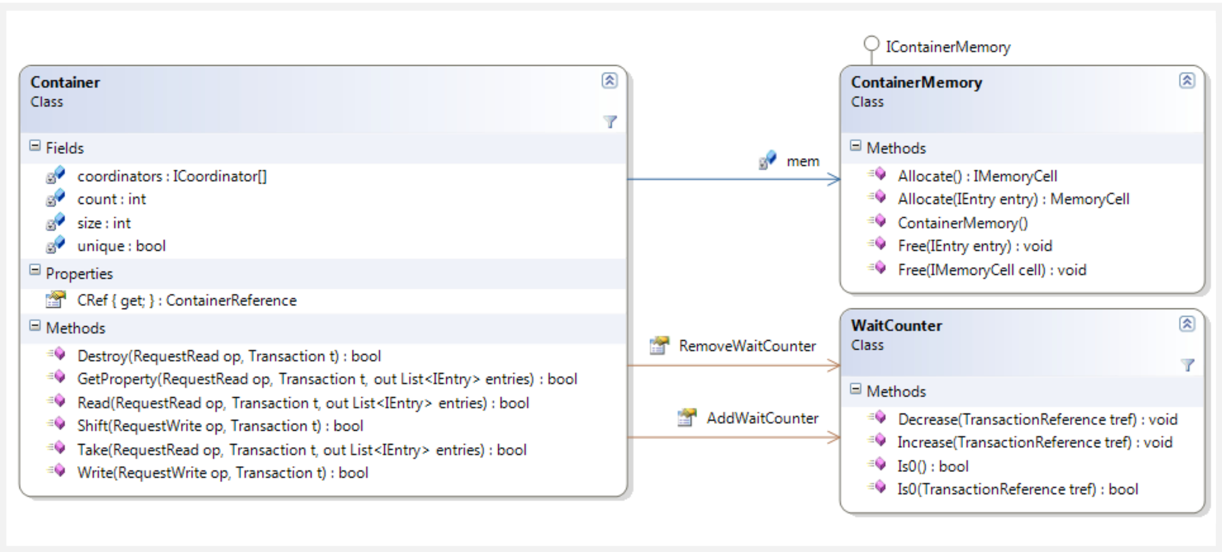


Figure 35: Class diagram of the kernel's container classes.

4.2.7.2 The Container Memory

Underlying each container is a *container memory*. This container memory is the place where the container stores all its data. To store single pieces of data, it provides *memory cells*. This is implemented in the *ContainerMemory* and *MemoryCell* classes. Every time an entry is written into the container, it is added to the container memory, and every time an entry is removed from the container it is also removed from the container memory. The container memory was originally implemented as a point for extending the kernel with persistency services, e.g. replacing the default container memory (that is storing data only in-memory) by one that stores the container's data in a file or database (every thinkable kind of storage mechanism could be implemented). When restarting the kernel after a system crash, the container memory could automatically reload its container's persisted data.

Although the container memory is present in the kernel, it is more of a test implementation and yet has no support for replaceability. The reason for this is that the support for persistency has not yet been important enough for putting an effort into it, and providing a reliable implementation of the container memory would in any case need to be tested thoroughly to its usefulness by implementing at least one persistency model with it. Because of its immatureness and because it is also still mostly of no use, the implementation of the container memory is not handled further here.

4.2.7.3 Coordination Types

According to the model, the implementation of the concrete coordination types is completely separated from the container itself. Every coordination type is implemented by two classes (not counting supporting classes), a *selector* that inherits from the abstract class *Selector* and a *coordinator* that implements the interface *ICoordinator*. The selector has exactly the task that is described in the model, to define which entries are read from a container and how an entry is coordinated within a container when it is written. The coordinator implements the logic of the coordination type and is used by a container to manage the data for this coordination type.

For example, the *fifo* coordination type consists of the two classes *FifoSelector* and *FifoCoordinator*. The FifoCoordinator manages all of the container's entries in a fifo (first in first out) order. When an entry is written into the container, the container informs the FifoCoordinator that a new entry has been written. When entries are read from the container (in a *fifo* coordinated container the entries would be read by using the FifoSelector), the container asks the FifoCoordinator which entries are read (since the FifoCoordinator manages the entries in fifo order, it knows perfectly which entry is the next to be read).

It has also been recognized during implementation, that, like the container, single coordination types can have special properties as well that are important to be read. A good example is the list of existing key values in a *key* coordinated container. Again, it would of course be possible to manage this information in the container's meta container, but doing that would surely decrease the *key coordinator's* performance. Because of that, the container's *GetProperty* Method has been extended to be used together with a selector. When e.g. the key values of a key coordinated container should be read, the *GetProperty* method just has to be used with the property's name and and the *KeySelector*.

For users of the kernel to write their own coordination type, they just have to implement a class pair of selector and coordinator. To make this as easy as possible, the classes and interfaces that are needed therefore have been separated from the kernel into a *contract* assembly. For more information about contracts and the *selector contract* in particular, see chapter 4.4. For a more detailed description about coordination types and their implementation in the kernel, see [12].

4.2.8 Transactions

Within the kernel, a transaction is represented by the *Transaction* class. The most important data of a transaction is its *transaction log*, which shows exactly which operations have been performed as part of this transaction (and in which order). The log contains three different kinds of information: First, information about entries that have been added to or removed from a container within this transaction, and second, information about the locks that have been acquired and are currently held by this transaction. The third kind of information that can be contained in the log is another transaction (which is then called a *child transaction*), because the transactions in the kernel are stored in a hierarchical structure (this is only used internally, to the outside the kernel doesn't support nested transactions).

All the items in the transaction log have one thing in common: They can be committed and rolled back. Because of that, the classes that represent these items all implement the *ITransactionLog* interface.

ITransactionLog	
Commit	Commits this log object.
Rollback	Rolls back this log object.

In addition to the *Transaction* class, the log is implemented with the *TransactionLog* and *TransactionLock* classes:

Transaction : ITransactionLog, ITransaction	
RootTx	The root transaction of this (child) transaction. This is needed for locking,

	because all locks must be acquired for the root transaction (otherwise two child transactions of the same root transaction could block themselves).
Log	The log of the transaction, as a List<ITransactionLog> object.
AddLog	Adds an ITransactionLog object to the log of this transaction.
Commit	Commits the transaction by calling *Commit* to all items in the log.
Rollback	Rolls back the transaction by calling *Rollback* to all items in the log.

TransactionLog : ITransactionLog	
CommitDelegate	A delegate method for defining which method must be called for commit.
RollbackDelegate	A delegate method for defining which method must be called for rollback.
Type	The type of action that is logged with this TransactionLog object. This is an information to help deciding what has to be done at commit or rollback. Possible types are e.g. *EntryAdd* and *EntryRemove*.
Entry	The entry that is concerned by this log object (e.g. added to or removed from a container).
Commit	Commits this log object by calling the CommitDelegate that has been set when creating the log object.
Rollback	Rolls back this log object by calling the RollbackDelegate that has been set when creating the log object.

TransactionLock : ITransactionLog	
LockReleaseDelegate	A delegate method for defining which method must be called for releasing the lock represented by this TransactionLock object.
Commit	Releases the lock by calling the defined LockReleaseDelegate method.
Rollback	Releases the lock by calling the defined LockReleaseDelegate method.

Both the *TransactionLog* and the *TransactionLock* class are using *delegates* (method pointers) for commit and rollback. This a very good way for letting these classes be as flexible as possible, and also letting the actual code for the commit and rollback actions stay where it is best placed: The class that is responsible for creating the log object. For example, the container class would define a *RollbackEntryAdd(TransactionLog t)* method. When an entry is added to the container, a TransactionLog is created and the *RollbackEntryAdd* method is used as RollbackDelegate in the log. As soon as the log's *Rollback* method is called (when the transaction is rolled back), it calls the delegate, which is the *RollbackEntryAdd* method. This way, the container itself defines how an action is committed or rolled back and all the logic stays within the *Container* class. The TransactionLock class works even easier, since it doesn't need two different possibilities for commit and rollback, because in both cases the lock just has to be released. Figure 36 shows an overview of the transaction classes in form of a class diagram.

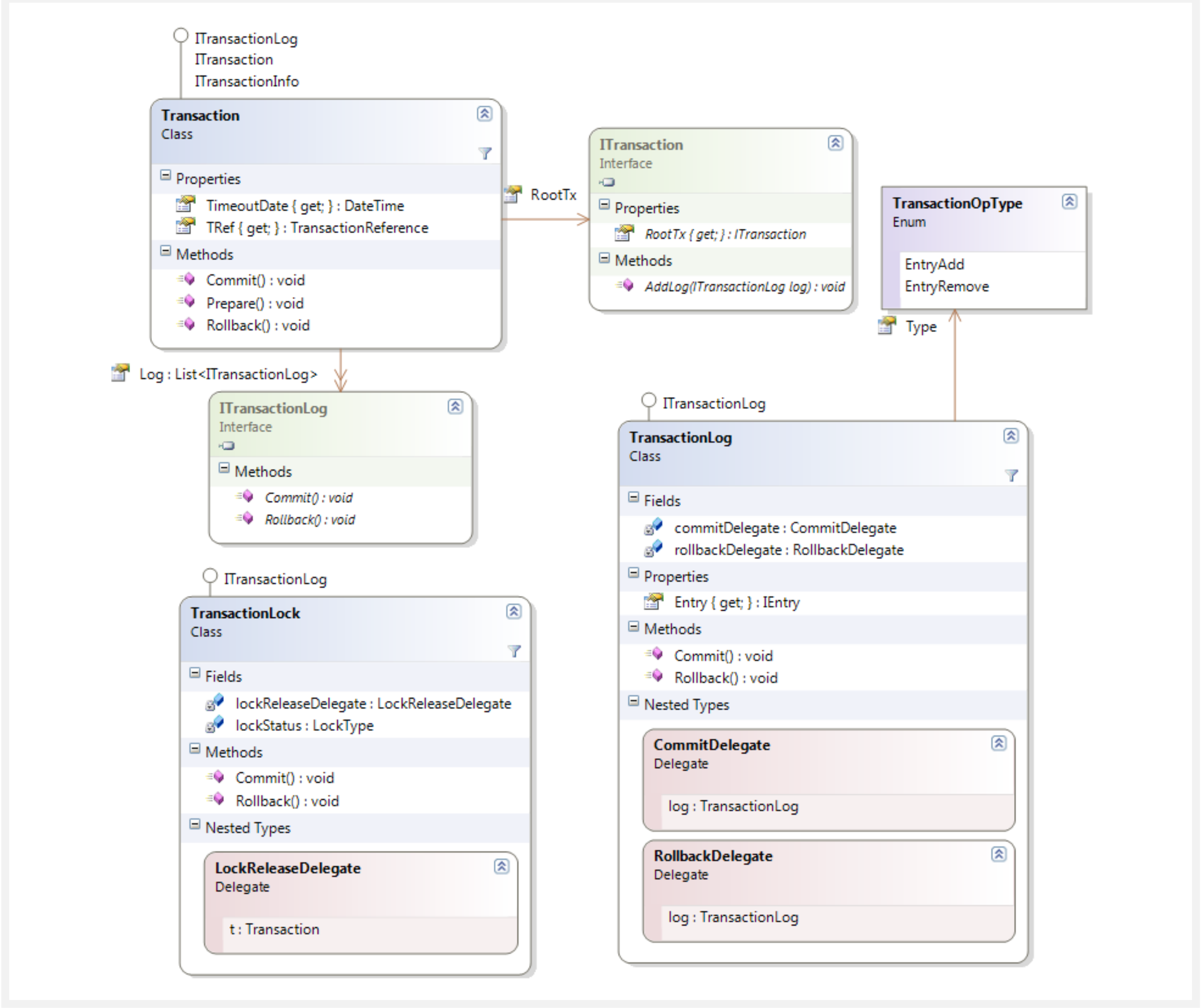


Figure 36: Class diagram of the kernel's transaction classes.

Figure 37 illustrates the use of transactions in the kernel with a write operation on a *fifo* coordinated container. The left side shows how items are added to the transaction log during an operation. Since the first thing when executing an operation on a container is acquiring a lock, a *TransactionLock* object that represents the lock is also the first item that is added to the log. Then, both the container and the coordinator (represented by the “Fifo Order” box) add items to the log for adding a new entry (the coordinator handles commit and rollback independently of the container, so it is free do decide by itself when something must be added to the log and what must be done at commit or rollback). The right side of the figure shows how the committing of a transaction (the same one as before) is handled. The transaction calls the *Commit* method of all its log items. This is done in the opposite order as the log items were added (which guarantees that the all the changes on a container are committed before the lock is released). By calling *Commit* on a log item, it calls the delegate method that has been set when adding the log item.

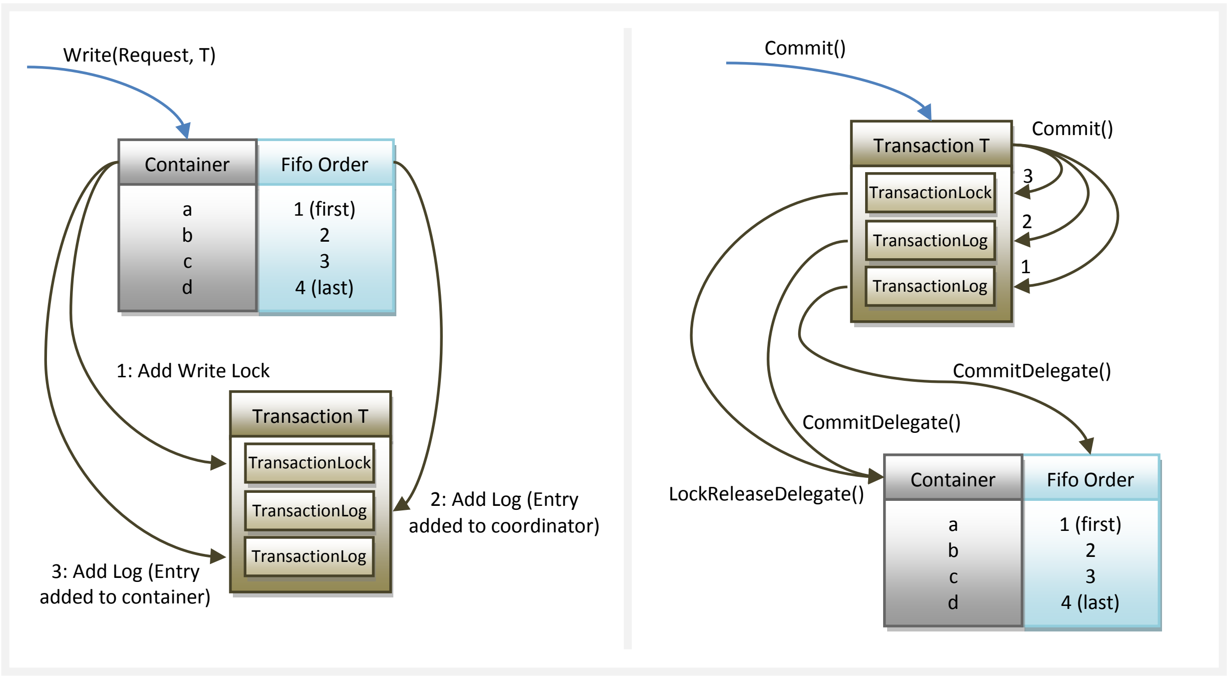


Figure 37: (1) Using the transaction log during a write operation, (2) committing a transaction.

The implementation of transactions is handled in more detail in [12].

4.2.9 Error Handling

For error handling in the kernel, exactly the exceptions have been implemented as defined for the core model (see chapter 3.4.6). The implementation only differs in the way that all exception names begin with "Xco" instead of "Xvsm". All exceptions are inherited from the base class *XcoException*.

One exception that is not part of the model and is special to the XcoSpaces implementation is the *XcoBindingException* which is thrown when there is an error binding a component to the kernel at runtime, like the thread dispatcher component. For more info on the dynamic binding of kernel components see chapter 4.3.4).

Since the processing of requests is done in a completely different thread than the thread of the user, an exception can of course not be thrown directly. Because of that, all exceptions that are thrown during processing in the core processor are caught and a response containing the thrown exception is created in such a case. In that way the core processor can also act as a control mechanism for ensuring that only exceptions of type *XcoException* are thrown by the kernel, and that if some other exception occurred a *FatalCoreException* is thrown instead.

If a response contains an exception, the embedded API rethrows it in the thread with which the XcoKernel's method was called. In that way, the processing components in the kernel can still throw their exceptions as they would normally do (by using the *throw* statement) without caring about who must receive it in the end, and the user gets the exceptions the way s/he would expect it.

4.2.10 Logging

One thing that is not really necessary mentioning in a model, but becomes very important when it comes to implementation, especially for a piece of software as complex as the kernel, is logging. It can be of great help for a developer to understand the internal processes of the software and to find

errors. But the requirement for logging to be really useful (and not on the contrary be an obstacle for the kernel) is of course that it is implemented right. This means that log messages must be produced by the kernel not only at points that are important for processing, but they must also be understandable and contain useful information that helps understanding the situation. Also, logging must be configurable to be turned on and off, so it doesn't decrease the kernel's performance. It would be very good to have different levels of logging, so that certain messages will only be displayed for debugging, while others (like error messages) can be configured to be displayed always. Finally, it would be best to be able to decide what is done with the output, e.g. if it is written in a file or directly displayed in the console.

Because logging mechanisms are already supported by the .Net Framework in form of the *System.Diagnostics.TraceSource* class as well as by free-to-use third party libraries like *log4net* [39], it was decided to use one of these solutions. Since they are specialized to the task of logging and have proven their worth, they are a far better solution than anything self-made. To rely not only on one particular logging solution, the implementation is hidden behind an interface which is one of the kernel's contracts (for more information about the *logging contract*, see chapter 4.4.2).

Within the kernel, the following different levels of logging are used:

- *Debug* for messages that give a very detailed output about the internal processing in the kernel and that are only of interest when debugging.
- *Information* for messages that don't contain very important information, but still something useful to know not only when debugging.
- *Warning* for things that happen in the kernel that are not unforeseen, but still could in some situations be the source for errors or be seen as a kernel error from outside the kernel, and are therefore a reason to warn the user about what has happened.
- *Error* for any errors that happen during processing in the kernel, while it is still guaranteed that the error only prevented the correct execution of a single operation and the kernel is still fully functional after that.
- *Fatal* for critical errors that could cause inconsistencies that affect the integrity of the whole kernel (and not just a single operation).

These levels are not a new invention for the kernel, they are rather common and implemented in a similar way by different logging frameworks (although not always implemented with the same names as above) and have thus already proven to be useful.

The *TraceSource* class has been chosen for the standard logging implementation because it doesn't provide less functionality compared to any third party library and is directly integrated in the .Net Framework and therefore doesn't need the use of any additional libraries. It also allows very simple configuration by using the application's *app.config* file. It is not only configurable which level of logging is used for output, but also where the output is going to, like to a file or directly to the console (or even both). The following code shows how the logging can be configured:

```
<system.diagnostics>
	<sources>
		<source name="XcoSpaces.Kernel" switchName="DefaultSwitch"/>
		<source name="XcoSpaces.Kernel.Core" switchName="DefaultSwitch"/>
		<source name="XcoSpaces.Kernel.Container" switchName="DefaultSwitch"/>
		<source name="XcoSpaces.Services.Kernel.Communication"
			switchName="DefaultSwitch"/>
```

```
        </sources>
        <switches>
              <add name="DefaultSwitch" value="Information" />
        </switches>
</system.diagnostics>
```
Code Example 2: Logging configuration with the TraceSource class.

The shown configuration splits in two sections, named *sources* and *switches*. In the *sources* section it is defined how the loggers are named in the kernel. Giving different loggers in the kernel different names has the advantage that the level of logging for different parts of the kernel can be set independently if needed. The name *XcoSpaces.Kernel* is used for logging in the API classes, like *XcoKernel* and *Notification*. The main classes of the internal kernel structure like *CoreProcessor*, *EventProcessor* and *TimeoutHandler* use *XcoSpaces.Kernel.Core* as name, while the *Container* class and its related classes (like the classes for locking) use *XcoSpaces.Kernel.Container*. Finally, there is an own name for the communication classes that is used for logging in the WCF communication service.

In the configuration shown above, the logging for all sources is kept at the same level. This is done by using a *switch* which is defined in the *switches* section. The value of the switch is set to *Information* which means that all log messages will be shown at this or a higher level. The switch could also be set to any other level, or to *Off* (meaning no logging is done at all).

This configuration of course just shows the most basic things that can be done, but it is already suitable for most situations. Since the shown configuration doesn't define where the logging output goes to, the standard output is used which is the console. Like with switches, it would be possible to define output types that are used by all sources, but also to define them just for a single source.

4.2.11 Documentation

Another thing not to be forgotten, especially since it has been identified as a core requirement, is *Documentation*. It is particularly important to see the documentation directly associated to the kernel implementation, and not just as some loosely related documents. Since the kernel is a component used for programming, the documentation should help programmers directly while implementing software that uses the kernel, by making use of the programming environment's features like auto-complete.

Visual studio supports that with the *xml documentation* feature. Classes, methods, properties and delegates can be documented by adding an xml documentation header to them. When compiling, this text can automatically be compiled into an xml file that represents the assembly's documentation. This means that in addition to the *XcoSpaces.Kernel.dll* file, which holds the kernel itself, a second file named *XcoSpaces.Kernel.xml* is created which holds the kernel's complete code documentation.

Using this xml file when programming with the kernel enables visual studio to display this documentation for classes, methods and so on, when using auto-complete or similar features.

4.3 Implementation based Kernel Design

For a complex piece of software like the kernel, it is important not only to give some thought about the architecture of the kernel (which has already been handled in detail by now), but also about how the software itself can be structured into single components (note that although these will also be called components further on, they don't necessarily have anything to do with the components of

the kernel architecture introduced earlier, because the point of view is a completely different one). The approach that has been chosen for that is called *contract first design* and is explained by Ralf Westphal in [40] and [41].

4.3.1 An Introduction to Contract First Design

Contract first design views software as consisting of *components*. A component can be seen as a logical unit consisting of one or more *assemblies*. This definition intentionally leaves out anything about what should be included in a component and how software should be split into components. What's important is that components encapsulate functionality that makes sense to be separated from other components. All further definitions could just get in the way, since that mainly depends on the piece of software itself.

Now, when using components there are two basic kinds of them: *Client components* (or consumers) and *service components* (or producers). A client component uses one or more service components to do its work.

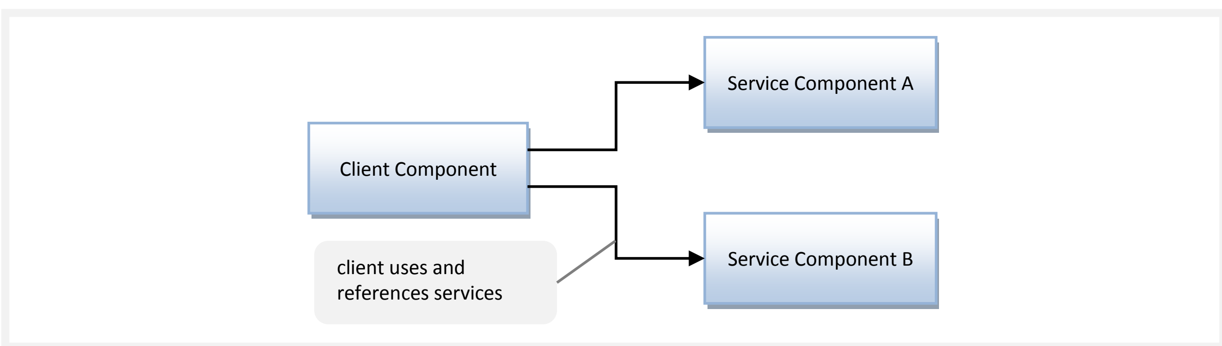


Figure 38: Client component using service components by direct reference.

Figure 38 shows how this is done by developers most of the time. Two visual studio projects are set up, one for the client component and one for the service component, and then a reference to the service component is added at the client component (because this is necessary to instantiate classes of the service component). In code, using a service component within the client component would then typically look like that:

```
ServiceComponentA a = new ServiceComponentA();
a.DoSomething();
```

Code Example 3: Using a service component in the traditional way.

But there are some clear disadvantages when doing it that way. First, to implement the client component the service component already needs to exist, otherwise it could not be referenced in visual studio. Developers are forced to develop applications strictly bottom-up, and that's bad for productivity. Second, testing the client component always requires the real service component. This could not only make testing more time-consuming if the service component takes a long time to execute, but also more complex as errors could not only occur in the client component but also in the service component. For testing the client component it would be much better to be able to replace the service component with a mock-up, but because the service component is instantiated and referenced directly by the client component this is not possible.

Contract first design overcomes these problems by changing the relationship between client and service components like shown in Figure 39: The client component no more references the service

components directly, but only so called *contracts*. A contract describes the services of a service component in terms of interfaces and other data types independent of any service implementation. All contracts are defined before the service components themselves are implemented, therefore the name *contract first* design. Because the client does not reference any service components directly any more, the components are completely independent of each other. This principle of decoupling components that are directly dependent on each other is also known as the *dependency-inversion principle* [42].

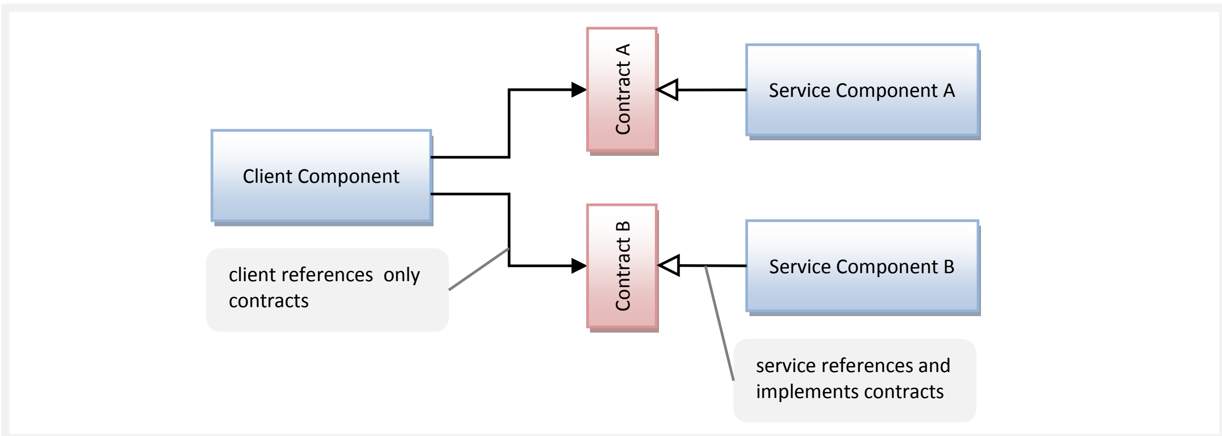


Figure 39: Prevent direct references by the use of contracts.

This way both problems from above can be solved: Since the client no more directly references to the service component, they can be developed independent of each other in any order, or also in parallel (only the contracts have to be present first). When testing, the service component can now easily be replaced by a mock-up that simply implements the given contract, there isn't any more need for testing with the real service component.

There is only one problem with this design: Since the client doesn't know the service component any more, it cannot instantiate any service classes directly any more. But it is of course inevitable that the client is still able to instantiate a service that implements the given contract. So if there is no static binding, how can that be done? The answer to the problem is that the binding happens dynamically at runtime, by using a *microkernel*. This is a small piece of software that is able to create an instance directly from an interface by using an implementation class that is bound to this interface dynamically at runtime, similar to the J2EE *service locator* pattern [43] or the *dependency injection* in the Spring framework [17]. The following code example shows how a service component is typically used with contract first design, without directly needing the service component implementation. The microkernel knows which class to instantiate depending on the given interface (assuming in this example that the class *ServiceComponentA* implements the *IServiceA* interface):

```
IServiceA a = Microkernel.GetInstance<IServiceA>();
a.DoSomething();
```

Code Example 4: Using a service component in contract first design.

How such a microkernel is implemented in the XcoSpaces kernel and how exactly it works is handled in chapter 4.3.4.

4.3.2 Contract First Design in the Kernel

The benefits of this design especially for the kernel can be easily shown with an example. Two components that are easy to be identified are the kernel itself (we just call the kernel's central component kernel) and the communication service. The kernel acts as the client component, it uses the communication service (the service component) for remote communication.

Figure 40: The kernel directly referencing to the communication service.

Figure 40 shows the situation without using contract first design, with the kernel referencing directly to the communication service. This is bad not only because of the reasons already mentioned above (no independent development, not individually testable), but it can also not fulfill two very important requirements of the communication service: First, the kernel should have a default communication service. Second, the communication service should be completely replaceable. But with the given solution the communication service would not be replaceable at all. This changes when adding a contract to the picture, as seen in Figure 41.

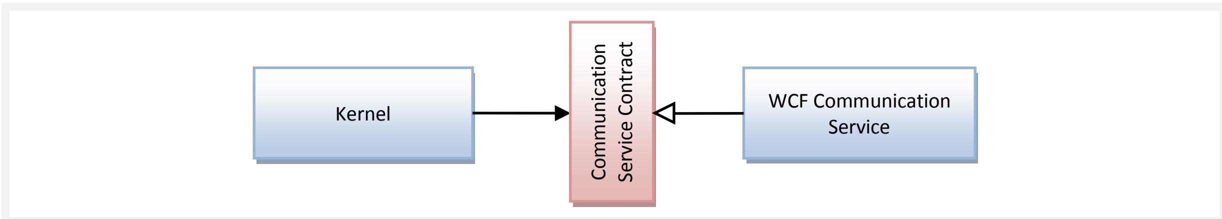


Figure 41: Direct reference between kernel and communication service removed by a contract.

The figure shows the use of a communication service contract. The contract is implemented by the WCF communication service, which is the default communication service. It can simply be made the default communication service by configuring the microkernel so that when the kernel instantiates the communication service by using the microkernel it gets an instance of the WCF communication service. Also replacing the component is very easy. Since the WCF communication service is not referenced by any other component, it can be easily replaced by any other communication service, just by reconfiguring the binding in the microkernel. Especially important is that the component can be replaced at runtime, no recompilation is ever necessary. So the choice of design fits excellent to the extensibility of the kernel.

4.3.3 The Kernel Component Structure

The kernel's component structure, based on contract first design, is shown in Figure 42. Components are named according to their classes' main namespaces. Contracts are shown in red while normal components are blue. Like in the example earlier, the black filled arrows show that another component is referenced, while the white filled arrows show that a contract is referenced and inherited (implemented). Every one of these components is built as an own visual studio project. This also means that in the end there is one assembly (dll library) for each component.

The main component is *XcoSpaces.Kernel*. It implements all of the functionality that has been explained in chapter 4.2, except the coordination types, remote communication, the core processor's

thread dispatcher and the logging mechanisms. So, the XcoSpaces.Kernel component implements the complete kernel structure, including core containers, messages, core processor, event processor, timeout handler and embedded API.

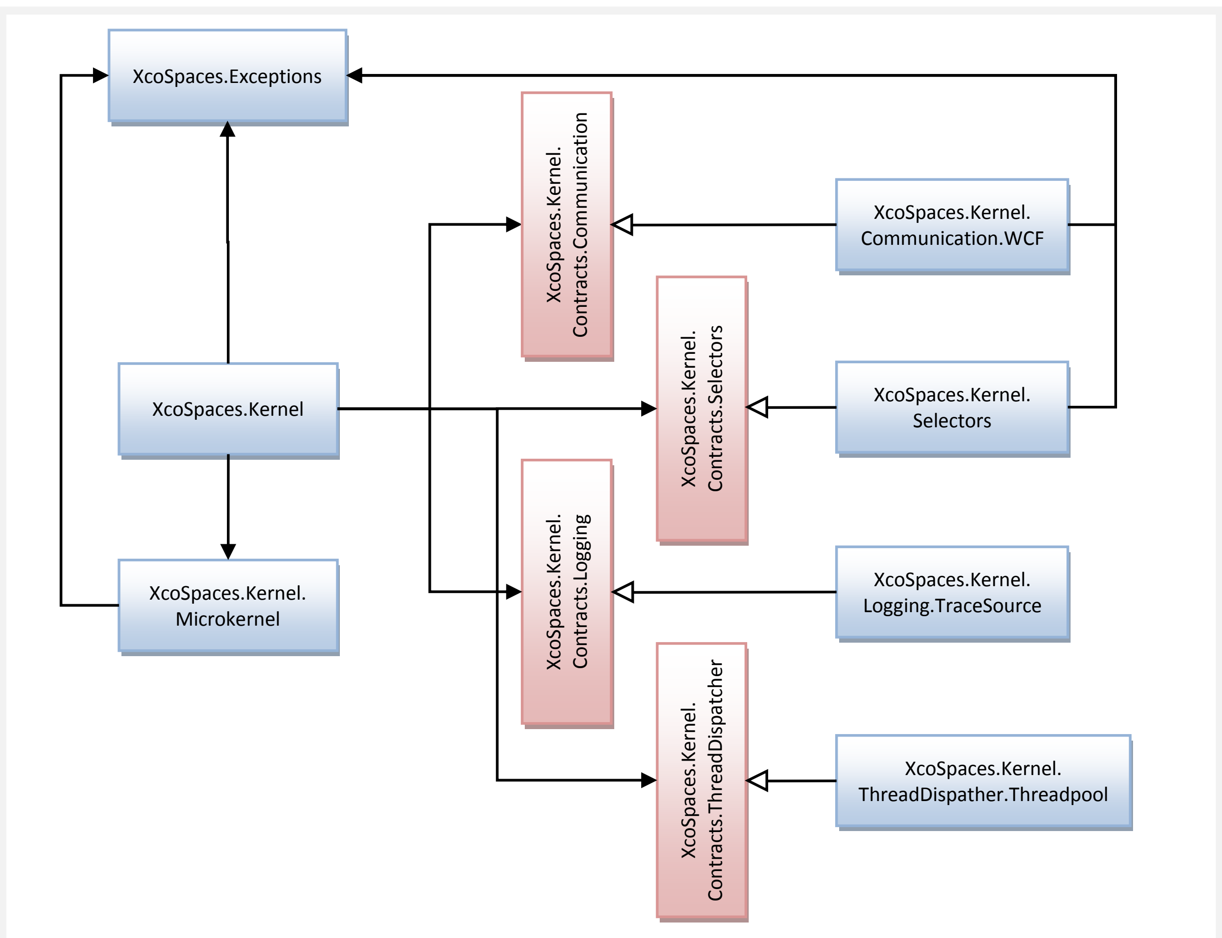


Figure 42: The component structure of the kernel.

The kernel directly references only to two components (meaning not contracts). One is the *XcoSpaces.Exceptions* component which contains all Exceptions that can be thrown in the kernel. The kernel's exceptions have been separated into an own component because they are also used by components other than the XcoSpaces.Kernel, so it is useful to separate them. The direct reference between these components (without contract) is clearly the better choice in this case, because the exceptions implement no logic at all and it wouldn't be of any benefit to have them replaceable (not even for testing). They could even be seen as a kind of "contract for errors".

The second component that is directly referenced is the *XcoSpaces.Kernel.Microkernel*. This is the component that allows the kernel to dynamically bind components that are not directly referenced at runtime. This functionality of the microkernel was implemented into an own component because it doesn't logically belong to the kernel's main functionality in any way and can also easily be separated without having any disadvantages, and it is of course helpful to remove some complexity from the *XcoSpaces.Kernel* component. The microkernel is explained in detail in the next chapter.

Despite of these two components, the kernel only references to contracts. Because of that, it doesn't know the components that are implementing these contracts and these components don't know the kernel (they only need to know and reference the contract they implement). The *Communication*

contract defines how a component that can be used by the kernel for remote communication has to look like. It is inherited by the *Communication.WCF* component that implements the WCF communication service. The *Selectors* contract defines the classes and interfaces for implementing coordination types. The *Selectors* implementation component implements seven different coordination types (there hasn't been seen a need to separate every coordination type into an own component). The *Logging* contract defines the interface that is used for logging in the kernel, it is implemented by the *Logging.TraceSource* component. Finally, the *ThreadDispatcher* contract defines the interface for the core processor's thread dispatcher. Its default implementation is present in the *ThreadDispatcher.Threadpool* component.

Since the contracts (and the components that implement them) are not only important inside the kernel but also provide powerful ways to extend the kernel, they are described more detailed in an own chapter (see chapter 4.4).

4.3.4 The Microkernel

The microkernel is the component of the kernel that enables the dynamic binding of the components that are not directly referenced to the *XcoSpaces.Kernel* component at runtime. Therefore, the microkernel must be able to store bindings from an interface to a class and instantiate the class for a certain interface as long as a binding exists for it. To be able to configure these bindings, the microkernel must be able to load them from a configuration file. The design of the microkernel is kept similar to the one Ralf Westphal introduces in [41], because Ralf's microkernel already fulfills all of the mentioned requirements.

The binding configuration is loaded from both the application configuration file (*app.config*) and an embedded configuration file (a file that is directly embedded into the assembly dll). The embedded configuration file holds a standard configuration (the bindings that should be used by standard if nothing else is configured for a certain interface in the *app.config*), e.g. for the communication service contract, the standard implementation would be the WCF communication service. In that way, it is no must to define bindings in the *app.config*, only bindings have to be defined that are different from the default binding. The configuration is stored in form of an xml (which is the best solution since the app.config is an xml file and using an xml format also helps providing a clear and simple structure). Every binding is defined by a pair of interface and implementation in the following form:

```
<add
   interface="interfacename,assemblyname"
   implementation="classname,assemblyname"
/>
```

Code Example 5: Definition of a binding in the microkernel configuration.

The *interfacename* must be the complete name of an interface (including the full namespace name), the *classname* the full name of a class. The class must implement the interface and must not be abstract. It must also have an empty constructor, so that the microkernel can create instances of it. All these requirements are checked by the microkernel by the time the binding is loaded. For both interface and class it is also necessary to define an *assemblyname*, which is the name of the assembly file without ".dll" (e.g. for the XcoSpaces.Kernel.dll assembly file, the name would just be XcoSpaces.Kernel). This is necessary for finding the class/interface by reflection as soon as it is not in the executing assembly.

The following code example shows the standard binding configuration of the kernel. The interface of a contract assembly is bound to the class of the assembly that implements the standard for this contract. The definitions include logging, thread dispatcher and communication service. For selectors no binding is needed because the kernel doesn't need to know the selectors implementation, it only works with the classes and interfaces of the selector contract (it is not necessary for a container in the kernel to know the implementation of selector and coordinator that it uses, no direct instantiation is done).

```
<XcoSpaces.ConfigServices>
  <add
    interface="XcoSpaces.Kernel.Contracts.Logging.ILogger,
      XcoSpaces.Kernel.Contracts.Logging"
    implementation="XcoSpaces.Kernel.Logging.TraceSource.Logger,
      XcoSpaces.Kernel.Logging.TraceSource"
  />
  <add
    interface="XcoSpaces.Kernel.Contracts.ThreadDispatcher.IThreadDispatcher,
      XcoSpaces.Kernel.Contracts.ThreadDispatcher"
    implementation="XcoSpaces.Kernel.ThreadDispatcher.ThreadPool.ThreadDispatcher,
      XcoSpaces.Kernel.ThreadDispatcher.ThreadPool"
  />
  <add
    interface="XcoSpaces.Kernel.Contracts.Communication.IXcoCommunicationService,
      XcoSpaces.Kernel.Contracts.Communication"
    implementation="XcoSpaces.Kernel.Communication.WCF.XcoWCFCommunicationService,
      XcoSpaces.Kernel.Communication.WCF"
  />
</XcoSpaces.ConfigServices>
```

Code Example 6: The kernel's standard binding configuration.

To load the configuration from the *app.config* file, the microkernel makes use of the *IConfigurationSectionHandler* interface for implementing a custom *configuration section handler*. Since the application configuration file can contain much more (and unrelated) information than just the binding configuration, it is divided into *sections*. The .Net Framework supports the easy loading of such configuration settings by letting users write their own configuration section handlers. When a configuration section handler is defined to be able to read a certain section in the configuration, it will always be used automatically whenever this section is read. The microkernel takes advantage of that with the *ServiceConfigSectionHandler* class that implements the interface mentioned above. This enables the microkernel to read the configuration with nearly no effort.

The main class of the microkernel is the *DynamicBinder*. It loads and stores the binding configuration and can be used to create an instance from an interface type for which a binding exists. It implements the *singleton* pattern [44], which guarantees that there is only a single instance of the *DynamicBinder* that can be accessed over its *Instance* property. This way, it is not needed for the kernel to keep an instance of the class and hand it over to any of the kernel's classes that need it.

DynamicBinder	
Instance	Gets a singleton instance of the DynamicBinder.
LoadBindingsFromConfig	Loads the bindings from the *app.config* and the embedded configuration file, and controls if the bindings are valid. Bindings defined in the *app.config* override bindings from the embedded configuration file.
CreateInstance<TInterface>	Creates an instance from the generic interface type given to the method. An instance of the class is returned that is defined in the binding configuration for this interface. An exception is thrown if no

	binding is configured for the given interface.

In the way the *CreateInstance* method is defined, it even provides type-safety to the outside because it is assured that exactly the given generic type is returned. Directly at startup, the kernel calls the *LoadBindingsFromConfig* method. As soon as that is done, it can easily use the microkernel to instantiate any of the bound interfaces. E.g. for the thread dispatcher the call would look like that:

```
IThreadDispatcher threadDispatcher =
   Microkernel.DynamicBinder.Instance.CreateInstance<IThreadDispatcher>();
```

Code Example 7: Using the DynamicBinder to create an instance for the thread dispatcher.

The microkernel could even be of good use for layers above the kernel that also want/need dynamic bindings to other components, the bindings therefore would just need to be defined in the *app.config* additionally to the kernel's bindings. Because of the singleton access the *DynamicBinder* class could be used just like within the kernel.

4.3.5 Component Deployment

When developing software it is also important to think about deployment. It is important to know if the software has special requirements to the system it is running on, or which steps have to be taken in order to get it running. As a light-weight piece of software, deployment for the kernel should best be as easy as copying a single file.

One disadvantage of contract-first design is that by splitting the kernel into many small components, it is also split into many single assembly files. Additionally, because for every assembly file there is also a documentation xml file, this makes a whole of 22 files for the kernel alone. A number of files that large is clearly not a good thing for deployment, since it is not only much more difficult to overlook so many files, but it also forces developers to reference a whole bunch of different libraries in their projects instead of only a single one to use the kernel.

To counteract this problem, an additional program is used for deployment named *ILMerge* [45]. ILMerge allows merging a set of assemblies into a single one, and can also do the same with the documentation files. In this way, the kernel can be deployed with only two files (assembly and documentation), and the component oriented structure that has already proven its great usefulness for development can still be used without any disadvantages. An illustration of the deployment with ILMerge is shown in Figure 43.

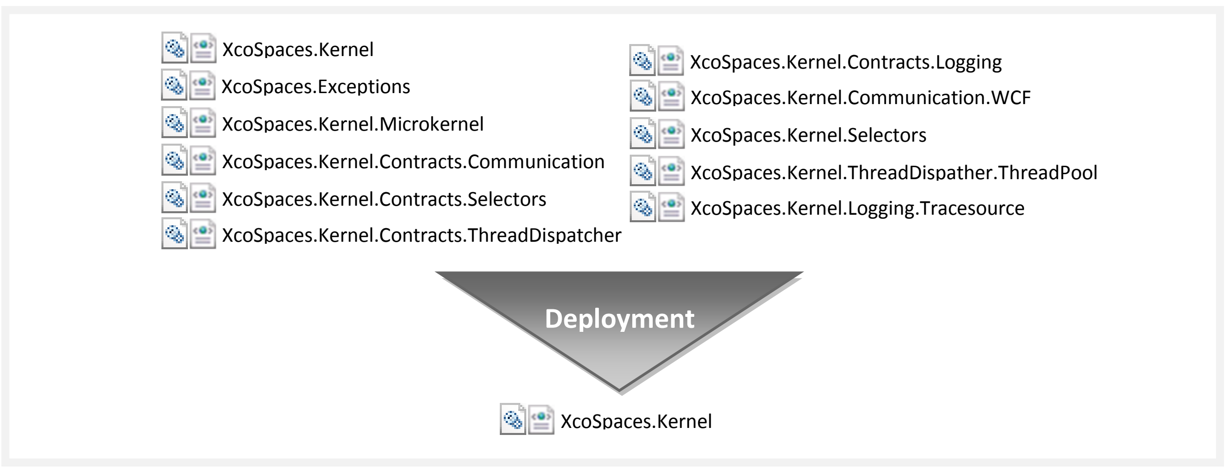


Figure 43: Merging the kernel components for deployment.

The use of ILMerge for the kernel brings even another powerful possibility: The features of ILMerge and the kernel's microkernel together could be used to easily compile different versions of the kernel with different configurations. For example, while the standard version of the kernel uses the *Threadpool* thread dispatcher component, another version could be merged together using the *CCR* thread dispatcher library instead with an appropriately altered embedded binding configuration file.

4.4 Contracts

The kernel's contracts define how components have to look like that can be bound dynamically into the kernel. Next to aspects, they can be seen as the kernel's main points of extensibility. The description of the contracts always shows a class diagram as a general overview of which classes, interfaces and delegates are part of the contract and then describes all important points of the contract in more detail.

4.4.1 Selectors

The selectors contract contains all classes and interfaces that are needed to implement coordination types (in other words, pairs of *selector* and *coordinator*). Coordination types must support all operations that can be done on a container (*read*, *take*, *destroy*, *write*, *shift* and reading properties), as well as provide full support for transactions (commit and rollback). Furthermore, when implementing a coordination type it can be chosen if the coordination type supports locking on entry level or container level. The kernel currently contains seven coordination types (implementations of this contract). For uniform class names, the names of the selector/coordinator classes always begin with the name of the coordination type and end with *Selector*/*Coordinator*, e.g. for *fifo* coordination the classes are named *FifoSelector* and *FifoCoordinator*.

The following description only shows the most important aspects of the selectors contract and implementations. For a more detailed description, see [12].

4.4.1.1 Contract

The following class diagram shows an overview of the Selectors contract:

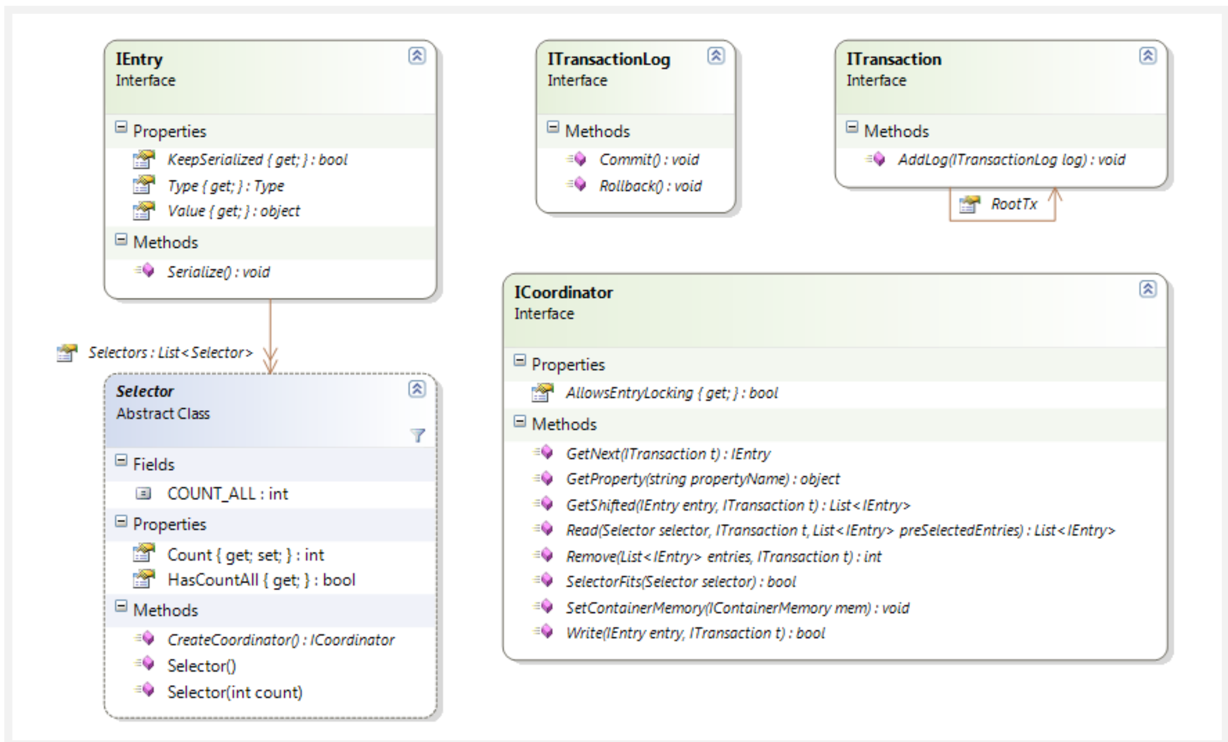


Figure 44: Class diagram of the Selectors contract.

Most important in the selectors contract are the abstract class *Selector* and the interface *ICoordinator*:

Selector	
Count	The number of entries that should be read (only when used in a read operation). A read operation will block until the defined number of entries is available.
COUNT_ALL	Constant to be used for the *Count* value, when all available entries should be returned. Using this also means that the read operation will not block if no entries are available, but return an empty list of entries.
CreateCoordinator	Creates a coordinator (instance of *ICoordinator*) for this selector that can be used to coordinate the entries in a container with the coordination type represented by this selector.

ICoordinator	
AllowsEntryLocking	True if the coordinator allows locking at entry level, false if it only allows locking at container level.
Read	Called when a read operation (*read*, *take* or *destroy*) is executed on the container, returns the list of entries that are read by a given selector. If the container has more than one coordination type, the *Read* methods of all its coordination types are called successively, in which case every coordinator's *Read* method has to take into account the entries that were pre-selected by the coordinator before.
Remove	Removes a list of entries from the coordinator.
Write	Adds one entry to the coordinator. Requires that the entry contains all coordination information that is needed by this coordinator to add it (in form of selectors).
GetNext	Gets the next entry that would be chosen by this coordinator to be removed when the container is full. This is used to remove entries from the container

	for shift operations.
GetShifted	Gets a list of entries that must be removed from the container so that the given entry can be added. This is used for shift operations, to determine if any entries in the container prevent the new entry from being written (e.g. in a *key* coordinated container, this would return an entry with the same key as the one that should be written, if such an entry is already in the container).
GetProperty	Gets a certain property from the coordinator. If the given property name is unknown an exception is thrown.
SelectorFits	Returns true if the given *Selector* fits to this coordinator (e.g. in a *FifoCoordinator*, this would return true for a *FifoSelector*, otherwise false.

The container uses the methods of its coordinators to do all of its operations. The way how a coordinator is structured makes it possible to let the container take away as much complexity from the coordinator as possible. For example, the coordinator is never responsible for dealing with locking, it just has to tell the container if it allows entry locking or not. Understanding how the container uses a coordinator is easier with a small example: E.g. in a *take* operation, the container would first call the coordinator's *Read* method. Then it would see if a lock can be acquired for the selected entries. If this is successful, the container would finally call the coordinator's *Remove* method, remove the entries from the *container memory* and then return it.

Since the coordinator must also support transactions, it must be able to recognize when a transaction is committed or rolled back and then be able to take certain actions (e.g. an entry that has been removed in a *take* operation must be added again to restore the state from before if the operation's transaction is rolled back). Therefore the coordinator must be able to use the transaction log. Because of that, the contract contains the interfaces *ITransaction* and *ITransactionLog*:

ITransaction	
RootTx	The root transaction of this (child) transaction. The coordinator can need this information when it implements entry locking to optimize its selection of entries for a certain transaction.
AddLog	Adds an ITransactionLog object to the log of this transaction.

ITransactionLog	
Commit	Commits this log object.
Rollback	Rolls back this log object.

With the *ITransactionLog* interface, every coordinator can just implement its own transaction log class and is by that completely free of any predefined implementations of the kernel.

4.4.1.2 Implementation: Fifo

The first coordination type implementation is *fifo*, which orders the container's entries like in a queue. It is an *implicit order* coordination type, meaning the order in which the entries are written and read cannot be changed, the entry to be read next is always the one that has been written first. There are many possible situations where *fifo* is of use, like in a producer-consumer scenario for communication between producers and consumers.

4.4.1.3 Implementation: Lifo

Like *fifo*, *lifo* is an *implicit order* coordination type. It manages the order of the container's entries like in a stack, so the next entry to be read is always the one that has been written last. Its usage situations are similar to *fifo*, but lifo would e.g. be preferred if newer messages should be taken by consumers before the older ones.

4.4.1.4 Implementation: Key

With the *key* coordination type, the entries are coordinated like in a hashtable. It could also be compared with the *Dictionary* class in the .Net Framework. It is an *explicit order* coordination type, because how an entry is coordinated in the container is decided on basis of the key that has been provided when writing the entry and this information never changes until the entry is removed from the container. The implementation supports genericity, meaning the type of the key can be chosen (e.g. it could be a string or an integer). *Key* is the only coordination type that supports entry locking, this is because for most other coordination types entry locking is either very difficult or completely impossible to implement because any change to the container's entries has an effect on the whole container. In a *key* coordinated container, changes are always limited to a single entry, so entry locking is implementable rather easy compared to the other coordination types.

There are many different possibilities to use key coordination, for example when data in the space needs to be accessed fast by a unique name. A key coordinated container can serve very well as single point of storage for an application's data within the space (it enables the application to use only one named container and stored the references to all other needed containers there, instead of needing to name all containers separately and thereby increasing the risk of using the same name as another application).

4.4.1.5 Implementation: Label

The *label* coordination is similar to *key*. Other than a key, a label is not unique, so multiple entries can be written into a container using the same label. So, *label* coordination can be seen like a hashtable that allows attaching multiple values to a single key. Also similar to *key*, *label* supports genericity for defining the type of the label.

Label coordination can serve very well as a supporting coordination type. For example, used together with fifo it would be possible to make a specialized producer/consumer scenario where some consumers take all entries from the container, but some only take entries labeled with a certain value (like the "local" and "remote" flagging in the core model's request and response containers).

4.4.1.6 Implementation: Vector

The *vector* coordination type manages entries like in a linked list. It is named after Java's *Vector* class. It is kind of a hybrid between implicit and explicit order, because entries can be inserted at an explicit index, but the index can change while they are in the container when other entries are inserted to or removed from a lower index than the one of the entry. The importance of this coordination type is clear: It can be used for everything that a *List*<> is used, for every task where something needs to be stored in a list and the single entries need to be accessible and/or be inserted to certain positions (if that would not be the case and only the first or last entry needed to be accessible, *fifo* or *lifo* would be the better choice).

4.4.1.7 Implementation: List

The *list* coordination type can be seen as an "improved" version to *vector*. It acts exactly like *vector*, but additionally supports the overwriting of an entry in the container at a certain index with *shift*. It is therefore more similar to .Net's *List*<> class. It was decided not to just implement this improvement into the *vector* coordination type to keep the functionality of *vector* the same as in the Java implementation of XVSM (for not loosing interoperability). This coordination type is therefore unique to the XcoSpaces kernel.

4.4.1.8 Implementation: Linda

Linda coordination enables a container to be used like a classical tuple space. In addition to the *LindaSelector* and *LindaCoordinator* classes, the implementation consists of the *ILindaMatchable* interface. The value of entries written to the container must implement this interface. When reading entries from the container, the coordinator uses this interface's *Matches* method to find entries that match the query object that must be contained in the selector that is used for reading. *Linda* coordination comes to use when it is necessary to select entries from the container by their content.

4.4.2 Logging

The logging contract is used in the kernel for logging information that helps understanding the kernel's internal processes and finding errors. The basics about the logging functionality in the kernel have already been discussed in chapter 4.2.10.

4.4.2.1 Contract

The contract consists of only one interface, named *ILogger*. It provides methods to log messages for every different level of logging:

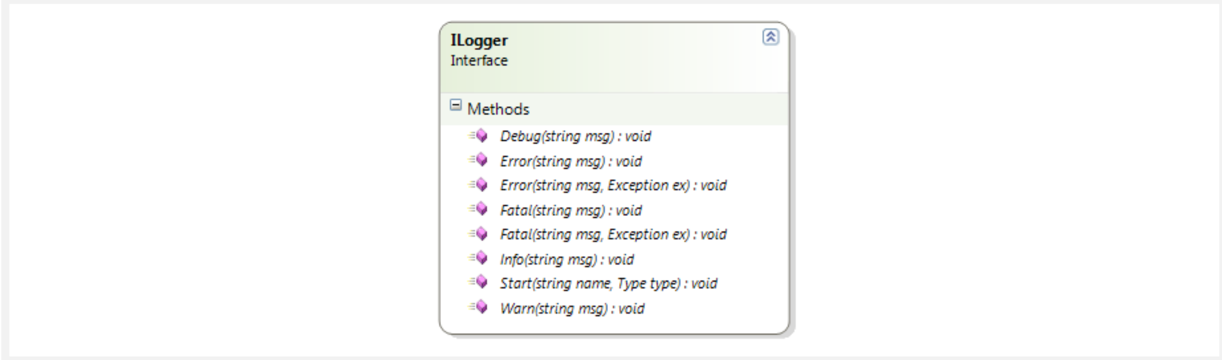


Figure 45: Class diagram of the Logging contract.

ILogger	
Start	Initializes the logger. The method takes a name and a class type as parameters which allow the logger to configure itself properly depending on for which purpose it is used.
Debug	Logs a debug message.
Info	Logs an info message.
Warn	Logs a warning message.
Error	Logs an error message. Optionally takes an exception as second parameter.
Fatal	Logs a fatal error message. Optionally takes an exception as second parameter.

The kernel just uses these methods and doesn't care which level of logging is currently set. The logger itself decides if a log message has to be logged or if it can just be ignored because the current logging level is higher than the level of the log message.

4.4.2.2 Implementation: TraceSource

The logging contract has only one implementation. This implementation uses the *System.Diagnostics.TraceSource* class of the .Net Framework. The name parameter of the logger's *Start* method is used as source name, which again can be used to configure a certain logging level in the *app.config* file. For the logging levels, a subset of values of the *TraceEventType* enum is used which can easily be mapped to the levels that have been defined for the kernel. So, the logger implementation stays extremely simple, using only a single line of code for implementing each of the *ILogger* interface's methods, and is yet very well configurable because of being able to define all logging levels in the application configuration file. Every other important aspect of the TraceSource implementation has already been explained earlier and will thus not be handled further here.

4.4.3 ThreadDispatcher

The thread dispatcher is used in the core processor to process requests in multiple processor threads concurrently. Its main tasks are giving an incoming request to a processor thread that is currently free, or queue incoming requests as long as all processor threads are occupied, and managing the core processor's threads in general (meaning the thread dispatcher has to handle things like the creation of new threads if needed). Details on how the thread dispatcher is integrated into the core processor have been handled in chapter 4.2.5.

4.4.3.1 Contract

The following diagram shows the classes that are part of the ThreadDispatcher contract:

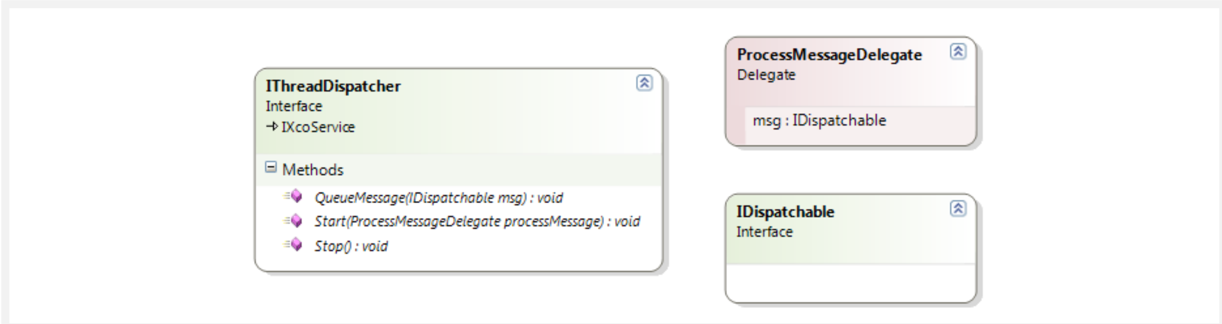


Figure 46: Class diagram of the ThreadDispatcher contract.

To not make the thread dispatcher dependent of the kernel's request messages (which would make it much more difficult to separate a clear contract from the kernel), the contract defines the *IDispatchable* interface for dispatchable messages. This (method- and property-less) interface is implemented by the *RequestMessage* class. Other than that, the contract defines the *IThreadDispatcher* interface which must be implemented by every thread dispatcher class:

IThreadDispatcher	
Start	Initializes the thread dispatcher and acquires all needed resources (like threads that need to be created).
Stop	Stops the thread dispatcher and frees all acquired resources.
QueueMessage	Takes an *IDispatchable* object as parameter. The given messages is given to

	the next free thread for processing (or queued up if no thread is currently free).

Additionally, the thread dispatcher of course also has to know what to do exactly with the messages that have to be processed. It therefore just uses a delegate method that is called in the thread that processes the message:

ProcessMessageDelegate
Delegate defining the method to be called for a message to be processed. Takes an *IDispatchable* as parameter.

The thread dispatcher's *Start* method takes such a delegate as parameter and the thread dispatcher then simply calls it in an own thread whenever processing a new message, with the message to be processed as parameter.

4.4.3.2 Implementation: Threadpool

The Threadpool implementation is the standard implementation of the kernel, using the *System.Threading.Threadpool* class of the .Net Framework. The advantage of this implementation is that it doesn't use any third party libraries, is very simple and has proven to be reliable. Since the *Threadpool* class allows no direct instantiation and only to use threads of the "standard" thread pool that applies to the whole application, no resources need to be acquired, all the resources being used are already there (so the *Start* and *Stop* methods practically do nothing). But this is also the downside of the this implementation, with the *Threadpool* class it is not possible to create an own pool of threads just for the core processor, which also implies that it relies on the rest of the application how many threads are currently free for the core processor to use. For processing a message, the thread dispatcher simply uses the *ThreadPool.QueueUserWorkItem* method which takes the method to be called and that method's parameters as input.

4.4.3.3 Implementation: CCR

The second thread dispatcher implementation uses the *CCR* [38] to start and manage its threads. Since it is not the standard implementation, it is not a fixed part of the kernel. In contrast to the *Threadpool* implementation, the *CCR* allows to start up a thread pool that is completely separated from the other threads of the kernel and overlying applications. This is done by creating a *Dispatcher* object, which is also clearly an advantage to the static *Threadpool* since it allows further configuration, like defining which type of object can be processed by the created threads. In the end, the *CCR* implementation shows a slightly better performance when having to deal with many concurrent requests at once (but the difference quickly becomes unnoticeable with less frequent requests).

The reasons for making the *Threadpool* implementation the kernel's standard thread dispatcher are that no additional library is required and that the CCR library is not allowed to just be bundled together with the kernel for distribution because of the license agreement under which it is published.

4.4.4 CommunicationService

The communication service is used by the kernel to communicate with other kernels. It is responsible for sending and receiving requests and responses and for dealing with all communication related

tasks like how to reach another kernel, providing reliability and handling communication errors. It is also responsible for message serialization and deserialization.

The following description only shows the most important aspects of the communication service contract and implementation. For a more detailed description, see [12].

4.4.4.1 Contract

The following class diagram shows the classes of the CommunicationService contract:

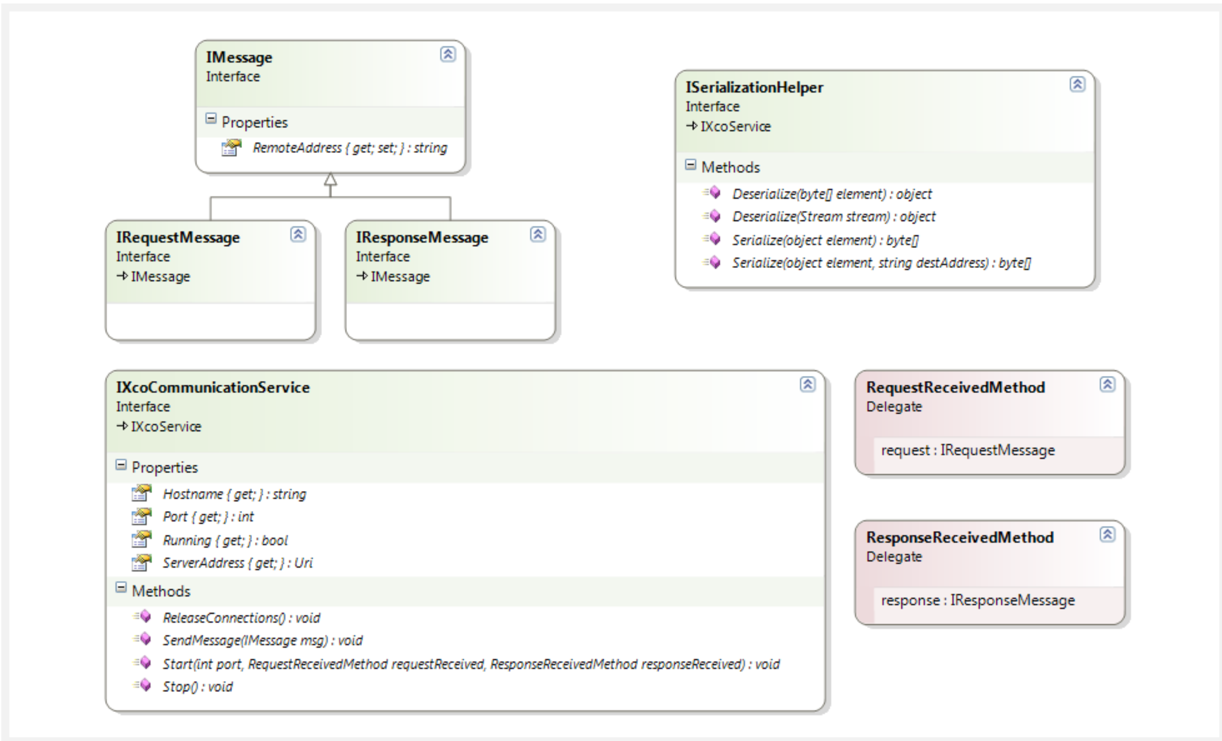


Figure 47: Class diagram of the CommunicationService contract.

The contract defines the *IXcoCommunicationService* interface that has to be implemented by all communication services. By working with delegates, the communication service doesn't need to know anything about where it gives received messages to, it just needs to call a certain method whenever it receives either a request or response message. Additionally to the type of message, the communication service needs to know to which address a message has to be sent to. To implement this functionality, the contract defines the three interfaces *IMessage*, *IRequestMessage* (which is implemented by the kernel's *RequestMessage* class) and *IResponseMessage* (which is implemented by the kernel's *IResponseMessage* class).

IXcoCommunicationService	
RequestReceivedMethod	Definition for the method to be called when a request (IRequestMessage) is received from a remote kernel.
ResponseReceivedMethod	Definition for the method to be called when a response (IResponseMessage) is received from a remote kernel.
Start	Starts the communication service at the given port, also takes two method delegates as parameters for the methods to be called when receiving requests or responses.
Stop	Stops the communication service.
SendMessage	Sends an IMessage to the given address.

IMessage	
RemoteAddress	The address to where the message should be sent, or the address from where the message has been received.

The *IRequestMessage* and *IResponseMessage* interfaces both inherit the *IMessage* interface. Since they are only needed for the communication service to distinguish between requests and responses, they both don't have any methods or properties.

4.4.4.2 Implementation: WCF

The standard communication service implementation uses the *Windows Communication Foundation* (WCF) [37] for handling all communication issues, which has been chosen because it is easy to implement and at the same time highly configurable. If not further configured, the WCF communication service uses TCP for communicating. For better performance, a connection to another kernel is stored and reused once it has been opened (because it has shown that opening and closing connections is very time consuming and slows communication significantly). Only when a connection is not used for a certain amount of time it is closed. For message serialization, the WCF implementation simply uses binary serialization.

5 Aspects in the XcoSpaces Kernel

5.1 Introduction

The most powerful extensibility feature of the core is the support for aspects. Aspects are more or less pieces of code that can be added at certain points in the core which are called *insertion points* (or *ipoints*). As soon as an aspect has been added at a certain point it will be executed whenever the execution of an operation in the core reaches that point. As is has been said earlier, there is one ipoint before and one after every operation, because of which these are called *pre*- and *post*-ipoints.

One important thing to guarantee as much flexibility as possible is the separation of aspects for a single container (called *local* or *container aspects*) and aspects for the whole space (called *global* or *space aspects*). This makes it possible to either change the behavior of only a container or of the whole space, depending on which kind of aspect is used.

It is of course also possible to add more than one aspect to the same ipoint, in which case the aspects are executed sequentially in the order that they were added. It can be very important to think about this order when adding aspects to the space since aspects concerning things like security could be needed to be executed before all other ones so the integrity of the core cannot be harmed.

For the aspect to be able to influence an operation running in the core (and also for the core to understand the outcome of an aspect call) there are four different return types that the aspect can use:

- **Ok**
  The execution of the operation can continue as normal.
- **Skip**
  The operation, as well as all succeeding pre- or post- aspects, should be skipped (but not throw an error). If used at a pre-ipoint the execution skips all succeeding pre aspects and goes directly to the execution of the post aspects. This can be used to implement aspects that actually completely replace the logic of a whole operation. If used in a post aspect, *Skip* has no impact on the operation itself (because its execution has already been finished) and only skips all succeeding post aspects.
- **Reschedule**
  The operation should be cancelled and immediately rescheduled. All succeeding pre- or post-aspects are skipped like with *Skip*.
- **Error**
  The operation should result in an error (defined by the aspect). This can be used for any situation where the aspect doesn't want the operation to be processed successfully or in case of errors in the aspect itself. All succeeding pre- or post-aspects are skipped.

All important things about aspects in practice are explained in the next chapters. A detailed introduction of aspects in theory can be found in [13].

5.2 Kernel Implementation

The implementation of aspects into the core in theory has shortly been introduced in chapter 3.4.4. For the XcoSpaces kernel the most important points of the aspect model have stayed the same. Mostly the core processor structure only has to be refined a bit, taking into account both container and space aspects. This is shown in Figure 48. (See chapter 4.2.5 for an explanation of the core processor implementation.)

5.2.1 Core Processor

Because of the separation of *container* and *space* aspects, these are also separated into different managers in the core processor. Since the aspects of different containers don't have any relation to each other, every container has its own *container aspect manager*. For managing the space aspects there is a single *space aspect manager*. Every aspect manager (both the one for the space and those for the containers) manages an individual list of aspects for every ipoint. Also, for every ipoint the aspect manager provides a method that calls all these aspects (always in the order that they were added) and also manages the skipping of aspects (as needed by certain aspect results).

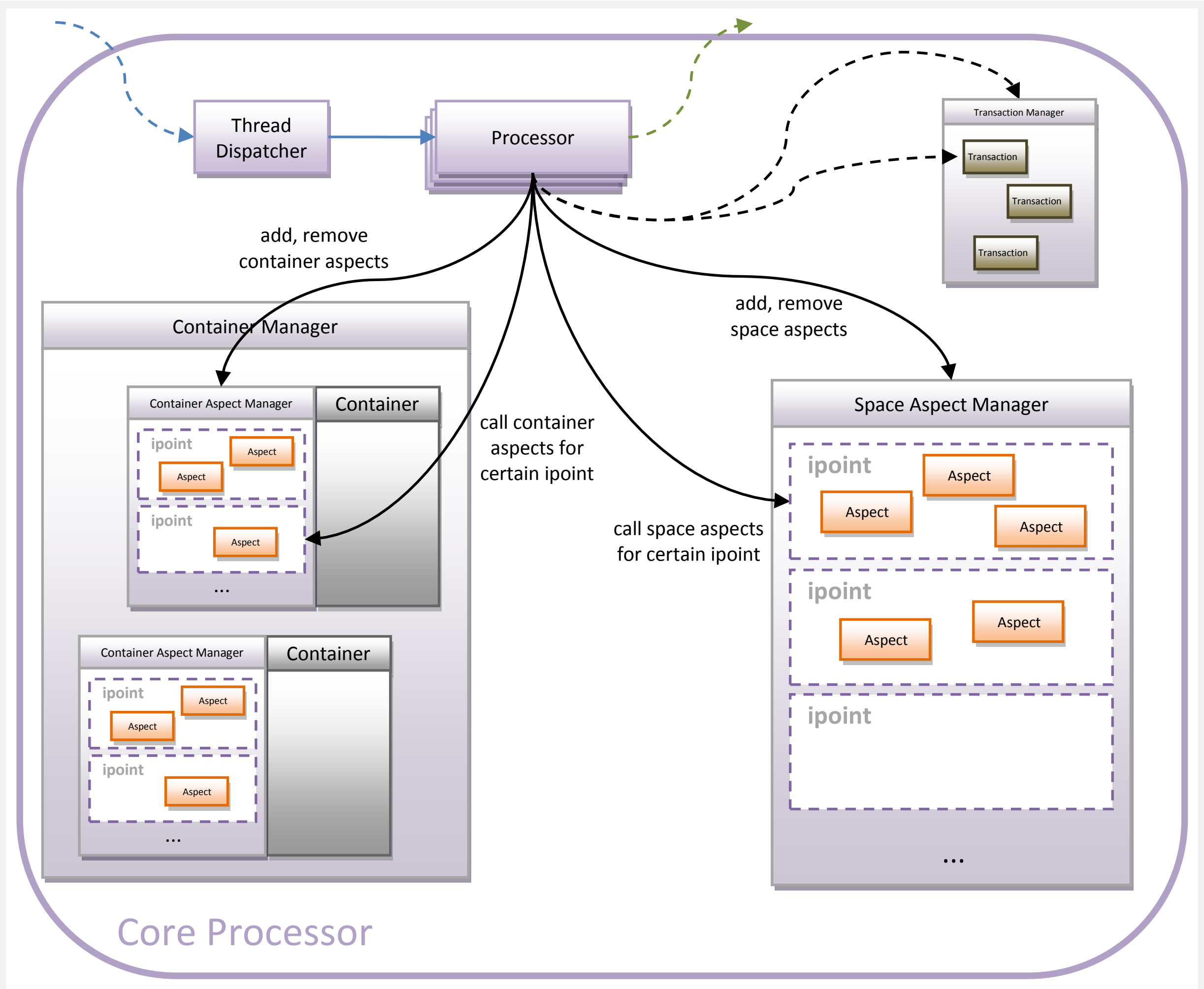


Figure 48: Aspects in the Core Processor.

When adding an aspect to an aspect manager a list of ipoints has to be provided that tells the aspect manager where it has to be added. The separation of the list of aspects into lists for every single

ipoint has two advantages: The user has more control over what the aspect does in the core. Although an aspect would normally always be added to the same ipoints, it could be that sometimes only a part of an aspect's functionality is wanted and therefore it doesn't need to (or even must not) be added to certain ipoints where it would normally be added to. The second advantage is that the performance can be controlled and optimized much better this way. If for every operation the processor would need to go through the list of all aspects and call all their methods (even if the methods do nothing but just return *Ok* as result), this would noticeably slow down operations even when they are not concerned by most of the aspects.

The following definition of the *ContainerAspectManager* class shows how the aspect managers are structured (only the methods for a few of the ipoints are listed):

ContainerAspectManager	
AddAspect	Adds an aspect to a given list of ipoints of the container.
RemoveAspect	Removes an aspect from a given list of ipoints of the container.
PreRead	Calls all aspects that need to be called before a read operation.
PostRead	Calls all aspects that need to be called after a read operation.
PreTake	Calls all aspects that need to be called before a take operation.
...	(there is one method for every ipoint)

The ipoints themselves are represented by the enumerations *ContainerIPoint* and *SpaceIPoint* (see Figure 49). So the list of ipoints for adding an aspect to the *ContainerAspectManager* is actually a list of *ContainerIPoint* values.

The results of the aspects calls (as described above) are represented by the *AspectResult* enumeration with the values *Ok*, *Skip* and *Reschedule*. In case of an error the aspect can simply throw an exception which then leads to the operation failing and giving the thrown exception back to the user. To be handled as an expected error, the thrown exception should always be inherited from the *XcoAspectException* class, otherwise the error will be seen as an unexpected error and lead to an exception thrown by the aspect manager telling that a certain aspect at a certain ipoint has failed to execute correctly.

5.2.2 XcoKernel

Since the XcoKernel provides possibilities for accessing all kernel functions it also needs to provide methods for the user to be able to manage the space's aspects. Therefore, some methods are added to the XcoKernel:

XcoKernel	
OperationContext	The operation context that is used for all of the kernel's operations.
AddSpaceAspect	Adds an aspect to a given list of ipoints of the space.
RemoveSpaceAspect	Removes an aspect from a given list of ipoints of the space.
AddContainerAspect	Adds an aspect to a given list of ipoints of the container identified by a given container reference.
RemoveContainerAspect	Removes an aspect from a given list of ipoints of the container identified by a given container reference.

The methods for adding and removing aspects are also implemented in a second variant where they take just an aspect as input parameter but no ipoints. These methods generate the list of ipoints from the given aspect on the fly, by looking at which methods the aspect overrides (the list is built from all ipoints from which the methods have been overridden). This is a convenient way for users to add aspects to the space (or remove them) with the default list of ipoints for this aspect without having to create the list of ipoints themselves.

Note that in the current version of the XcoSpaces kernel it is only possible to add aspects to the local space. So when a container aspect is tried to be added to a remote container, an exception is thrown. Adding aspects to a remote space by transferring an aspect object would cause some problems that cannot easily be solved: First, aspects would need to be made serializable or there would be some method needed to instantiate aspects directly at the space where they are needed but still set the variable properties of the aspect. Second, it would always be needed that not only the user that wants to add the aspect but also the remote kernel where the aspect should be added has loaded the class definition of this aspect. Adding an aspect unknown to the remote kernel would be impossible. Later versions are planned to support the adding of aspects to a remote space by the use of scripting ([13] handles this matter in more detail).

5.3 Implementing Aspects for the Kernel

An aspect can be implemented by inheriting from the abstract base class *ContainerAspect* (for implementing a container aspect) or *SpaceAspect* (for implementing a space aspect). Each of these abstract classes defines one method for every (space/container-) ipoint (pre read, post read, pre take, …). Such a method is then called by the aspect manager if the aspect has been added to the ipoint where this method belongs to. As input parameters the methods take all important parameters of the operation, e.g. directly before creating a container (post container create ipoint) the important parameters are the size, uniqueness and coordination types for the container to be created, directly after (pre container create ipoint) an important parameter is the container reference that will be given back to the user. Parameters which the aspect can have influence on are handed over as *ref* parameters so the aspect is able to directly change them. Other parameters that are currently available are handed to the aspect method for information purposes only.

The following class diagram gives an overview of the aspect classes in the kernel (ContainerAspect, SpaceAspect and their base class) and the enumerations that define the lists of ipoints in the kernel (ContainerIPoint and SpaceIPoint):

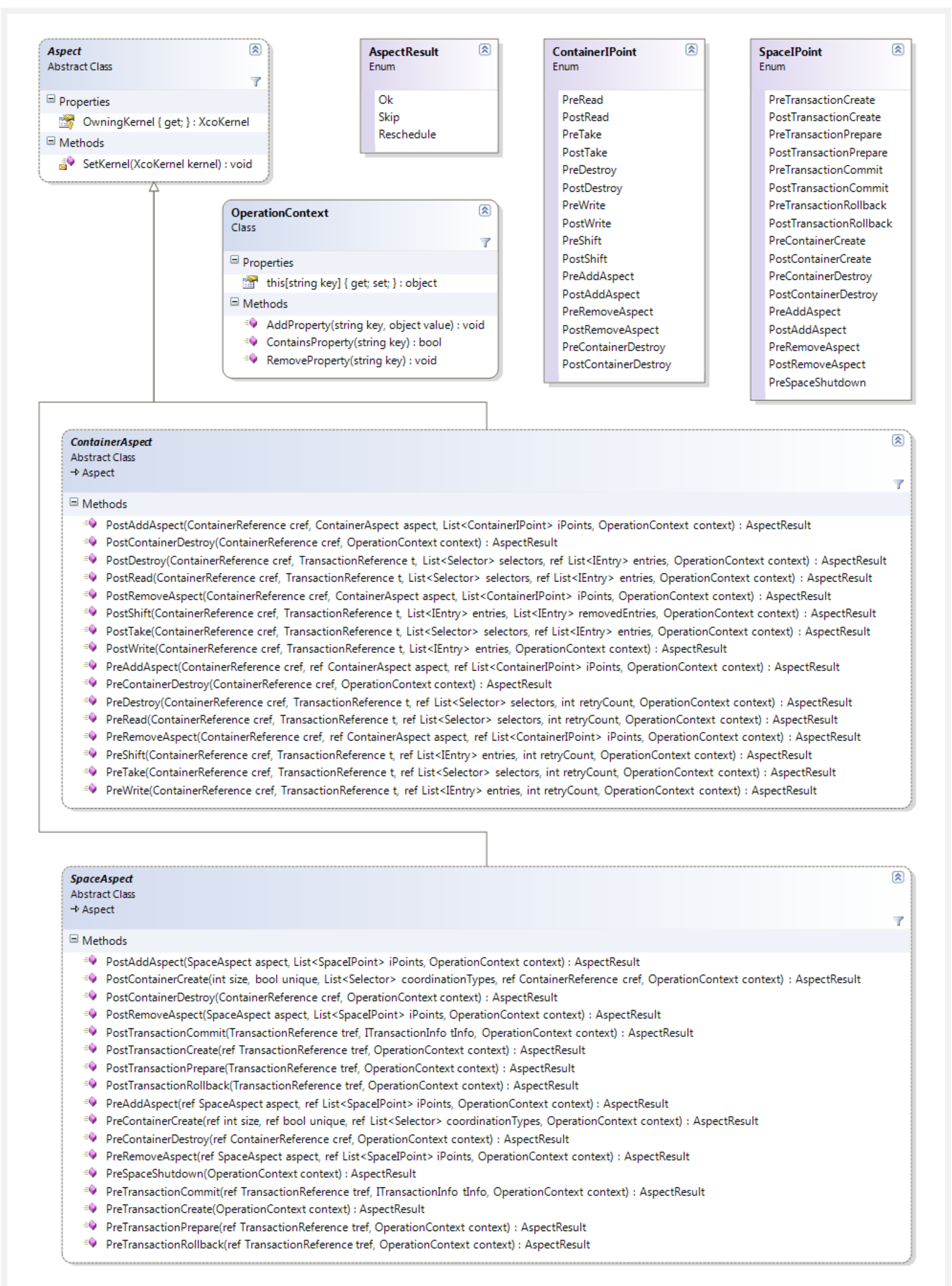


Figure 49: Class diagram of the kernel's aspect classes.

How the definition/implementation of a method within an aspect class looks like (by example of the pre/post container create ipoints) is also shown in the following code example. The

PostContainerCreate method still gets the size, uniqueness and coordination types as parameters but at this point they cannot be changed any more (because the container has already been created).

```
public virtual AspectResult PreContainerCreate(ref int size, ref bool unique,
       ref List<Selector> coordinationTypes, OperationContext context)
{
       return AspectResult.Ok;
}

public virtual AspectResult PostContainerCreate(int size, bool unique,
       List<Selector> coordinationTypes, ref ContainerReference cref,
       OperationContext context)
{
       return AspectResult.Ok;
}
```

Code Example 8: Methods for pre/post container create in the SpaceAspect class.

The code example also shows some more important details about aspects: The methods in the (abstract) *ContainerAspect* and *SpaceAspect* classes are not abstract, they are already implemented with a default behavior so that they do nothing and just return *AspectResult.Ok*. This makes the code for implementing an aspect a lot shorter because not all methods for all existing ipoints have to be overwritten, only the ones that are really needed by that aspect. (To be able to overwrite the methods, they are of course all marked with the *virtual* keyword).

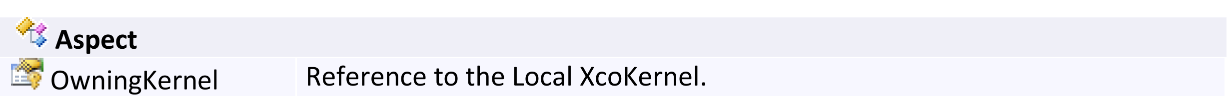
The second thing that is shown in the code example is that all aspect methods have an *OperationContext* object as parameter. Sometimes aspects will not only need the information they get delivered in a certain operation any way, but also some information about the *context* of the operation itself, e.g. information about which user has executed this operation. The aspect could use this information for example to allow certain operations only to certain users.

The user can store key-value pairs in the operation context with a string as key and any object as value. The aspect can then read these properties and use it, or it could even use the operation context to store values itself and other aspects that are called after this one can read these values.

OperationContext	
this	Array accessor for the properties of this OperationContext.
AddProperty	Adds a property to the OperationContext.
ContainsProperty	Checks if a key is contained in the OperationContext.
RemoveProperty	Removes a property from the OperationContext.

Another important thing is that an aspect must be able to access the space itself, because only then many of the things that are planned to be done with aspects can really be done. Aspects must e.g. be able to create and use containers, an example where this can be absolutely necessary even in simple cases is shown later.

Letting the aspect access the space is only possible when the aspect has a reference to the local kernel (the *XcoKernel* instance). This is done by a protected property. Because this property is similar in both the space and the container aspects it is part of the abstract class *Aspect* which both ContainerAspect and SpaceAspect inherit.



Aspect	
OwningKernel	Reference to the Local XcoKernel.

The *OwningKernel* property is automatically set by an *internal* method (not visible outside the XcoSpaces.Kernel assembly) by the time the aspect is added to the space, it can be accessed from inside the aspect but not changed.

How aspects can be used in practice and what possibilities they offer is best explained with an example.

5.4 Aspect Implementation Example

5.4.1 Introduction – The SimpleSecurityAspect

The aspect that is going to be used in this example implements a very simple security model and is therefore called *SimpleSecurityAspect*. The goal is to let only users do something within the space when they authenticate themselves with a valid username and password. If the user and password are valid, the space should behave completely normal, but if not an exception should be thrown telling the user that he is not allowed to access the space. At least the one who added the aspect to the space should also be able add new users, change the password of existing users and remove users while the space is running.

5.4.2 Implementation of the SimpleSecurityAspect

Since all of the space's possible operations should be secured, the *SimpleSecurityAspect* needs to be implemented as a *SpaceAspect*. The pre-ipoints are perfectly suitable for checking if the user is valid or not. If the user is not valid, an exception can be thrown and the operation will not be executed but return an error instead. Since this is an expected exception, it should be implemented as a subclass of *XcoAspectException*. We simply call this Exception *SimpleSecurityException*.

Of course an operation doesn't normally deliver a username and password. So a way is needed to transport the username and password to the kernel so that the aspect can read it when needed. The *OperationContext* introduced before is perfectly suitable for that. There just need to be two key names defined for username and password (like "simplesecurity.user" and "simplesecurity.password", so names will surely not interfere with those of any other values stored in the operation context). Before a user accesses the space he can add the username and password to his operation context and it will then be sent together with every operation.

Since it is enough that the person that adds the aspect can manage users (which implies that the aspect is added to this person's local space, since aspects cannot be added to a remote space), this can simply be done by methods of the aspect itself. So the aspect can simply provide *AddUser* and *RemoveUser* methods to do that.

To also secure access to containers a *SpaceAspect* is not enough. This can only be done with a *ContainerAspect*. Again, the aspect can just be registered at all of the container's pre ipoints and throw an exception if the authentication information is not valid. Because every container that is created should also be secure, the *SimpleSecurityAspect* has to add a container aspect to every new container directly after it is created. This also has the advantage that if it later should be possible to secure only certain containers in the space and leave other containers unsecured, the aspect can easily be changed to decide if it secures a container or not by the time the container is created just by adding the security aspect to it or not.

The following code example shows the implementation of the *SimpleSecurityAspect* class:

```
public class SimpleSecurityAspect : SpaceAspect
{
	public const string PROPERTY_USER = "simplesecurity.user";
	public const string PROPERTY_PASSW = "simplesecurity.pass";

	private Dictionary<string, string> users = new Dictionary<string, string>();

	public void AddUser(string user, string password)
	{
		users[user] = password;
	}

	public void RemoveUser(string user)
	{
		if (users.ContainsKey(user))
			users.Remove(user);
	}

	private void CheckUserValid(OperationContext context)
	{
		if (!context.ContainsProperty(PROPERTY_USER) ||
			!context.ContainsProperty(PROPERTY_PASSW))
			throw new SimpleSecurityException(
				"No authentication information could be found.");
		string user = (string)context[PROPERTY_USER];
		if (!(users.ContainsKey(user) &&
			users[user] == (string)context[PROPERTY_PASSW]))
			throw new SimpleSecurityException("Authentication not valid.");
	}

	public override AspectResult PreContainerCreate(ref int size,
		ref bool unique, ref List<Selector> coordinationTypes,
		OperationContext context)
	{
		CheckUserValid(context);
		return AspectResult.Ok;
	}

	...

	public override AspectResult PreTransactionRollback(
		ref TransactionReference tref, OperationContext context)
	{
		CheckUserValid(context);
		return AspectResult.Ok;
	}


	public override AspectResult PostContainerCreate(int size, bool unique,
		List<Selector> coordinationTypes, ref ContainerReference cref,
		OperationContext context)
	{
		this.OwningKernel.AddContainerAspect(
			cref, new SimpleSecurityContainerAspect(this));
		return AspectResult.Ok;
	}
}
```

Code Example 9: Implementation of the SimpleSecurityAspect.

The user management is simply done with the use of a dictionary that has the username as key and the password as value. The *AddUser* and *RemoveUser* methods allow adding, changing and removing users in this dictionary. The *CheckUserValid* method looks into an OperationContext object to see if it contains the needed username and password and checks if they are valid. The property names for

both values are defined as public constants. If the check leads to the authentication being invalid a *SimpleSecurityException* is thrown.

All the methods for the pre ipoints are now simply using the the *CheckUserValid* method to check the authentication information and then return *Ok* as result (because if no exception has been thrown everything is fine). The only method for a post ipoint that needs to be implemented is *PostContainerCreate* because directly after container creation an aspect needs to be added for securing the created container. This container aspect is implemented as an inner class named *SimpleSecurityContainerAspect*. Just like the aspect for the space, the container aspect implements the methods for all pre ipoints and there uses the *CheckUserValid* method to check the user authentication. The following code example shows its implementation:

```
private class SimpleSecurityContainerAspect : ContainerAspect
{
	private SimpleSecurityAspect securityAspect = null;

	public SimpleSecurityContainerAspect(SimpleSecurityAspect securityAspect)
	{
		this.securityAspect = securityAspect;
	}

	public override AspectResult PreDestroy(ContainerReference cref,
		TransactionReference t, ref List<Selector> selectors, int retryCount,
		OperationContext context)
	{
		securityAspect.CheckUserValid(context);
		return AspectResult.Ok;
	}

	public override AspectResult PreRead(ContainerReference cref,
		TransactionReference t, ref List<Selector> selectors, int retryCount,
		OperationContext context)
	{
		securityAspect.CheckUserValid(context);
		return AspectResult.Ok;
	}

	...

	public override AspectResult PreRemoveAspect(ContainerReference cref,
		ref ContainerAspect aspect, ref List<ContainerIPoint> iPoints,
		OperationContext context)
	{
		securityAspect.CheckUserValid(context);
		return AspectResult.Ok;
	}
}
```

Code Example 10: Implementation of the SimpleSecurityContainerAspect.

5.4.3 Using the SimpleSecurityAspect

With these preparations done, the *SimpleSecurityAspect* is ready to be used. The following code example shows how the aspect is added to the space and how the user authentication with the use of the *OperationContext* works:

```
using (XcoKernel kernel = new XcoKernel())
{
	//add the SimpleSecurityAspect to the space and create a test user
	SimpleSecurityAspect securityAspect = new SimpleSecurityAspect();
	securityAspect.AddUser("testuser", "testpassword");
	kernel.AddSpaceAspect(securityAspect);
```

```
        //accessing the space without authentication results in an error
        try
        {
                kernel.CreateContainer(null, -1, false, new FifoSelector());
        }
        catch (SimpleSecurityException ex)
        {
                Console.WriteLine("Result without authentication: " + ex.Message);
        }

        //after adding a correct authentication information to the kernel's
        //OperationContext, the space can be accessed
        kernel.OperationContext = new OperationContext();
        kernel.OperationContext.AddProperty(
                SimpleSecurityAspect.PROPERTY_USER, "testuser");
        kernel.OperationContext.AddProperty(
                SimpleSecurityAspect.PROPERTY_PASSW, "testpassword");
        ContainerReference cref = kernel.CreateContainer(
                null, -1, false, new FifoSelector());
        kernel.Write(cref, null, 1000, new Entry("test"));
}
```

Code Example 11: Using the SimpleSecurityAspect.

The example first shows how the aspect is added to the space and a user named *testuser* is added with the password *testpassword*. By using the *AddSpaceAspect* method with only the aspect as parameter, the aspect is automatically added to all ipoints for which it has implemented methods (which are all pre ipoints and the post container create ipoint). The following operation for creating a container at the same space results in a *SimpleSecurityException* because the kernel's operation context contains no username and password information. The exception is the one that is thrown by the aspect itself. Always when an aspect throws an exception it is transported back as response and then rethrown in the user's thread. After that the kernel's operation context is provided with a correct username and password and the kernel can then be accessed without problem.

5.4.4 More Ideas for the SimpleSecurityAspect

The given example of a security aspect of course only provides very limited functionality, but it should basically just show how easy it is to extend the space with aspects and alter the functionality of all space operations. It would be fairly easy to add some more functionality to the given aspect to advance it from just a nice little example to something that could already be interesting for using in a small application that just needs a basic possibility for user authentication:

- Information about the users could be extended by predefined roles, like administrators and standard users. Some operations could then only be allowed for certain user roles, e.g. while every user can read from and write to containers, only administrators are allowed to create and destroy containers.
- Instead of managing the list of users locally within the aspect, it could also be managed within a container in the space (which of course would need to be secured so that only administrators can change it). This would allow the list of users not only to be managed locally, but also from remote kernels, and a security aspect in a remote kernel would be able to use the same container for user authentication.
- When creating a container, a user could decide which users should have access to it, for example by providing a list of usernames in the operation context when executing the container create operation.

5.5 Notifications in the XcoSpaces Kernel

This chapter handles the implementation of notifications in the kernel which is based on aspects, and also gives a short introduction to notifications in theory.

5.5.1 Introduction

As discussed in chapter 3.2.2.2, one of the requirements for the core is the support for notifications. Basically, notifications are a way for the user of a space to automatically be informed when something has happened in the space, e.g. when an entry has been added to or removed from a certain container.

Such a notification could be implemented/used in many different flavors: The most basic form of notification would just be to inform the user that something has changed in the container (e.g. an entry has been added) and nothing more. The user would then need to take action himself (e.g. read the new entry from the container), but could on the contrary as well decide to just ignore what has happened. Another form of notification could not only inform the user but also directly deliver the entry/entries to the user that was/were part of the action. This could be useful when the user always wants to know which entries were involved in certain changes, e.g. which entries were removed from a container by *take* or *destroy*.

But there are many more things to think about when implementing notification behavior. The user should be able to choose on which actions he wants to be notified, e.g. only when entries are added or only when entries are removed. He could even be given the possibility to define that he only wants to be informed when the change on the container fulfills certain criteria, like only entries that are added to a key coordinated container having “x” or “y” as key value. More things to be decided could be: Must it be guaranteed that the user is informed about every event that occurs? Must the order of notifications be the same one as the order of events in the space that triggered them? What is the notification’s behavior in combination with transactions (e.g. should it fire on uncommitted changes)? All these questions are handled in detail in [13].

5.5.2 The Notification1 Definition

Since the implementation of notifications into the core has been identified as a requirement, but it would be impossible for one implementation to fulfill all possible flavors of notifications (it also wouldn’t make sense to put much energy into implementing all possible flavors when some of them are only needed very rarely), a certain subset of these flavors needs to be built for that, which is called the *Notification1* definition. This definition contains the features that have been identified as most important for notifications in the space.

5.5.2.1 General Behavior

The main purpose of a Notification1 is to inform the creator what has happened on a specified container. There are two important constraints Notification1s have to follow. First, the Notification1 mechanism is never allowed to miss any event that involves its container as target. The second one is that it just informs about the past and gives no information about the container’s content at the moment of firing (e.g. by the moment a notification fires because of a written entry, it is not guaranteed that this entry is still in the container). It can either just inform or return the entries that lead the Notification1 to fire.

5.5.2.2 Targets

Notification1s can have different targets, which decide when a Notification1 should fire. Those targets include *write*, *shift*, *read*, *take* and *destroy* operations on a specified container. For example, a Notification1 with the target *write* is supposed to fire whenever an entry has been written to that container. If the notification is supposed to return the affected entries, the written entries will be returned. Notification1 with the target *take* is supposed to fire whenever entries have been taken from that container, and should return the taken entries if it is supposed to. The same applies to the other operations.

Concerning the selection of certain events, the Notification1 is not needed to support any preselection. The use of *selectors* would be best suited to do that but would also bring some difficulties when certain coordination types like *vector* are involved.

5.5.2.3 Transactions

When an operation on a container is performed within a transaction, a Notification1 will fire only as soon as the transaction is committed (and not fire at all if the transaction is rolled back). This means that as soon as a transaction is committed, the notification immediately fires for all events of this transaction. It is important to mention that what is fired at that time is not based on what the transaction changed in the container overall, but exactly on what has been done within the transaction. E.g. if an entry has been written and then destroyed in the same transaction, the Notification1 (if it has both *write* and *destroy* as targets) will fire for both of these operations when the transaction is committed.

There is also the possibility to create a Notification1 "within" a certain transaction, in which case the notification immediately fires for all operations that are done in this transaction.

5.5.3 The Aspect Based Notification Implementation

As said at the beginning of this chapter, the implementation of aspects in the XcoSpaces kernel is based on aspects. It has been chosen to implement notifications with aspects because aspects provide all the possibilities needed to do that, and in this way no additional mechanisms have to be implemented directly into the kernel. Also, using aspects saves implementation time and also reduces the amount of additional code that needs to be tested for errors. The only drawback of implementing notifications with aspects is that it would of course be easier to optimize notifications for performance with a native implementation. But since aspects would of course needed to be optimized for performance themselves (otherwise they would not be suitable for all the problems that aspects are aimed to solve), it has been decided to use them anyway.

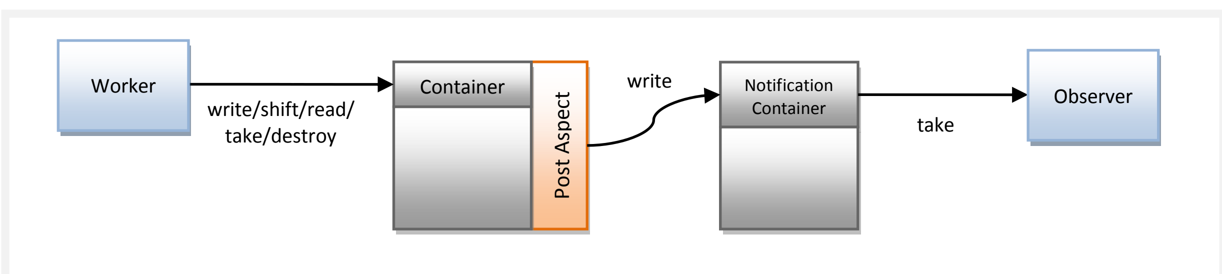


Figure 50: Notifications in the kernel implemented by aspects.

Figure 50 shows a in a simplified way how an aspect is used to implement a notification. There are two clients that use the space, one is the so called *worker* (because he executes operations on the displayed container), and the other one the *observer* that wants to get notified about every change in the container (thus further also called *observed* container), in other words the client that wants to use the notification. An aspect is added to the observed container that is called at the post ipoint of *write*, *shift*, *take* and *destroy* (all operations that change the content of the container). After a write or shift operation this aspect creates a new notification object and puts all written entries into it, after a take or destroy operation the aspect does the same, only with all the entries that have been taken/destroyed. This notification object is written to a so-called *notification container*, which is just a *fifo* coordinated container. The observer does nothing more than execute a blocking take on the notification container. Now, whenever something is changed in the observed container the aspect is triggered and the changes are written into the notification container. As soon as this happens the observer's blocking take operation is automatically woken up and the notification object which has just been written into the notification container is returned. After that the observer just has to do a take on the notification container again and wait for the next firing of the notification.

5.5.3.1 The Notification Aspect

The notification aspect is implemented in the *NotificationAspect* class, which is (of course) a subclass of *ContainerAspect*. It implements the methods for the post ipoints of *write*, *shift*, *read*, *take* and *destroy*. When the aspect is instantiated, the reference to the notification container is given to it. This container reference is then used by the aspect when one of its implemented post ipoint methods is called to write an object into the notification container that contains the entries that were the target of the operation and the kind of operation that occurred. For which operations a notification fires when it is added to a container depends on which ipoints the aspect is added to, e.g. if the notification should fire only for *write* and *shift*, the aspect would be added to the pre write and pre shift ipoints, but not pre read, take and destroy.

It must of course also be guaranteed that the aspect can easily be removed again as soon as the observer wants to cancel the notification. It is much easier for the observer to do that without even needing to directly know the aspect. Also, it should be guaranteed that as soon as the observed container is destroyed, the notification container that is now no more needed also gets destroyed automatically.

These two things are guaranteed by the aspect implementing the *PostContainerDestroy* method and being added it this ipoint to both the observed container and the notification container. When the observer wants to cancel the notification, he just needs to destroy the notification container. The aspect's *PostContainerDestroy* method is called and the aspect recognizes that its notification container was destroyed and can then removes itself from the observed container (by just calling the owning kernel's *RemoveContainerAspect(…)* method). On the other hand, if the aspect's *PostContainerDestroy* is called from the observed container, the aspect can immediately destroy the notification container because it is no more needed.

5.5.3.2 The Notification Class

Although the implementation of notifications is done by using aspects internally, the user should of course still get the feeling that he is dealing with *notifications* and not bother about anything concerning the use of aspects. To make the user experience notifications in the way it should be, it should also be prevented that the user himself has to open an own thread and use it for doing the

blocking take operation to the notification container. Therefore a notification presents itself to the user in form of the *Notification* class. When a user wants to create a notification on a certain container, he needs to use one of the following two methods in the *XcoKernel*:

XcoKernel	
CreateReadNotification	Creates a notification on a given container that has the operations *read*, *take* and/or *destroy* as targets. Returns a *Notification* object.
CreateWriteNotification	Creates a notification on a given container that has the operations *write* and/or *shift* as targets. Returns a *Notification* object.

When calling one of these two methods the XcoKernel creates a notification container and adds a *NotificationAspect* to the target container that writes into the newly created notification container. The kernel then creates a *Notification* object that knows the reference to the notification container and returns it. The Notification works with delegate methods to inform the user that something has happened.

Notification	
SetReadCallback	Sets a callback delegate method that should be called whenever the notification fires because of a *read*, *take* or *destroy* operation.
SetWriteCallback	Sets a callback delegate method that should be called whenever the notification fires because of a *write* or *shift* operation.
Start	Starts taking notification objects from the notification container in a separate thread with blocking take. Calls the read or write callback method as soon as a new notification object is received.
Stop	Stops the notification by destroying the notification container.

The callback methods have a list of entries and an operation type as parameters. As soon as the user calls *Start* the defined callback method will be called whenever the notification fires. A simple call to *Stop* destroys the notification container and thereby removes the *NotificationAspect* from the observed container.

6 Future Work

There are still many things to do concerning XVSM and its implementation XcoSpaces, starting from improvements on the current functionality to extending XcoSpaces with new features. The following list gives an overview of the most important future work:

- More work needs to be done to prove that the XVSM model goes into the right way. There exist certain ideas of improving the container structure and making it better usable and more understandable than it is now. These ideas will need to be compared and tested against the current model. Also, more prove is needed by extensive benchmarking of the implementations, to show if the XVSM core model is really as extensible as it claims to be (until now benchmarks have only been done within simple scenarios).
- These changes in the model also have an influence on XcoSpaces, being an XVSM implementation, and will thus need to be implemented in into the XcoSpaces kernel.
- The implementation of aspects aims to provide everything that is needed for extending the core with features like security, persistency and many more. Although simple aspects have already been implement for testing, many more aspects will need to be implemented to test if the given aspect functionality is really enough and suitable for everything that should be achieved with them.
- Also connected to the last point: The XcoSpaces kernel may already be in a rather mature stage, but the kernel alone only provides few features, much has been left out from the kernel to be later implemented with the use of aspects as so-called profiles. Such profiles need to be implemented in order to provide additional important functionality for the kernel.
- With the currently available communication services in XcoSpaces, the kernel still has problems in certain scenarios, especially when it comes to communicating over firewalls and NAT. New communication services need to be implemented (one that is using the *Jabber* protocol is already planned) to overcome these problems.
- Tools for administration, management and debugging of XcoSpaces need to be built that for example allow looking into the space and see the currently existing containers and their data, as well as allow general configuration of the space, e.g. how many threads are used in the core processor thread pool.

7 Conclusion

This document presented the architecture of the XVSM core, which is the central piece of software representing XVSM. This architecture tries to take into consideration all requirements that emerge from the XVSM model as good as possible. The document also introduced the XcoSpaces kernel and shows how it implements the XVSM model. In addition, the design of the kernel from an implementation point of view has been shown, with a special aim for introducing the contracts that are part of XcoSpaces and allow flexibly replacing several parts of the kernel.

A special focus has also been set on introducing the extensibility mechanisms of XcoSpaces called aspects, and to show how they work and that they really enable adding new behavior to the kernel and extending existing functionality. The SimpleSecurityAspect showed that this is absolutely possible.

The document also introduced a structure for classifying systems that implement (or are similar to) the shared data spaces paradigm, and surveyed a bunch of systems according to this structure. It became clear that creating a good classification structure is not an easy task, because taking all possible points of interest into account would make it too heavy, but concentrating only on certain points can easily make some systems look better than they are (and others look worse) when other points remain unmentioned.

The following table gives a short overview of the systems that were presented in the classification and compares them to XVSM. A star means that a certain function is supported (and no star that it isn't). Topics marked with *(R)* have a rating instead, with three stars being best and no stars worst.

	Blitz	GigaSpaces	LighTS	Corso	XVSM
Coordination Concepts					
Linda Coordination	*	*	*		*
Fifo Coordination	*	*			*
Key Coordination		*			*
Other Coordination Types				*	*
Meta Data (R)		*	*		***
Space Substructures (R)	*	*		**	***
Data Type Support (R)	**	***	**	*	***
Operations					
Basic Operations (R)	**	***	**	**	***
Events	*	*		*	*
Transactions	*	*		*	*
Extensibility					
Coordination (R)			**		***
Other Concepts (R)	*	*	*		***
Architecture					
Embedded System	(*)	(*)	*		*
Standalone System	*	*		*	(*)
Client/Server Based	*	*		*	
P2P Based					*
Partitioning/Repl./Caching (R)		***		**	*
API Support (R)	*	***			**
Security (R)		***		**	*

Judging from the results of the classification, the systems that came out to be outstanding above the others are GigaSpaces and XVSM. GigaSpaces is great concerning usability, support for different

coordination types and above all its extensible and scalable server architecture with support for all different kinds of replication and caching mechanisms. XVSM stands out even more when it comes to coordination, and has a great amount of extensibility.

This shows at least to a certain degree, that XVSM really does well in the points where its design aimed it to do. But it also shows what has already been mentioned in the future work chapter: Currently XVSM clearly falls behind other systems when it comes to functionality like replication, caching, persistency and security features. Although these functions may not be needed everywhere, they are clearly very important. Profiles, although already planned for supporting these features, are not yet implemented, and will need to be implemented not only to support all the needed features, but also to show if the introduced extensibility features, above all aspects, are really suitable for all of that.

8 Appendices

8.1 Figure Index

Figure 1: Classical forms of communication in a distributed system: (1) client-server communication and (2) direct communication. 10
Figure 2: Serverless communication with a space. 11
Figure 3: Classification Structure for Space Based Computing Systems. 16
Figure 4: A core that hosts a space with three containers. 26
Figure 5: A space with local and remote containers. 27
Figure 6: A container holding the elements a, b, c and d. 33
Figure 7: A Container that is coordinated by fifo coordination type. 33
Figure 8: A Selector for the coordination type *key* that is used to read / write an element with key "x1". 33
Figure 9: An entry with value "a" and no selectors. 34
Figure 10: An entry with value "a" and a selector that defines the key value "x1" for this entry, for a *key* coordinated container. 34
Figure 11: A container that is bounded to size 4. Because there are 4 entries in it, the container is full and cannot take any more Entries. 34
Figure 12: Reading an entry from a *fifo* coordinated container. 35
Figure 13: Taking an entry from a *fifo* coordinated container. 35
Figure 14: Destroying an entry in a container with *fifo* coordination. 36
Figure 15: An entry is written to a container with *fifo* coordination. 36
Figure 16: A new entry is shifted into a *key* coordinated container and replaces another one. 37
Figure 17: An entry is shifted into a bounded container that is already full, which causes another entry (a) to be removed. 37
Figure 18: A container and its meta container, containing several user-defined and non-user-defined values. 39
Figure 19: Basic core structure. 40
Figure 20: Standard scenario of a local method call in the embedded API. 43
Figure 21: Standard scenario for remote communication between two cores. 44
Figure 22: Structure of the core processor. 45
Figure 23: The core wait and event processing structure. 48
Figure 24: The core timeout handling structure. 49
Figure 25: Peer architecture. 52
Figure 26: Client architecture. 52
Figure 27: Standalone architecture. 53
Figure 28: Extending the core with profiles. 53
Figure 29: The XcoSpaces kernel structure. 56
Figure 30: Class diagram of the kernel's core container. 59
Figure 31: Class diagram of the kernel's request/response message classes. 60
Figure 32: Class diagram of the kernel's wait/event message classes. 63
Figure 33: Class diagram of the kernel's core processor classes. 64
Figure 34: The structure of the kernel's core processor. 65

Figure 35: Class diagram of the kernel's container classes. 69
Figure 36: Class diagram of the kernel's transaction classes. 72
Figure 37: (1) Using the transaction log during a write operation, (2) committing a transaction. 73
Figure 38: Client component using service components by direct reference. 76
Figure 39: Prevent direct references by the use of contracts. 77
Figure 40: The kernel directly referencing to the communication service. 78
Figure 41: Direct reference between kernel and communication service removed by a contract. 78
Figure 42: The component structure of the kernel. 79
Figure 43: Merging the kernel components for deployment. 82
Figure 44: Class diagram of the Selectors contract. 84
Figure 45: Class diagram of the Logging contract. 87
Figure 46: Class diagram of the ThreadDispatcher contract. 88
Figure 47: Class diagram of the CommunicationService contract. 90
Figure 48: Aspects in the Core Processor. 93
Figure 49: Class diagram of the kernel's aspect classes. 96
Figure 50: Notifications in the kernel implemented by aspects. 103

8.2 Code Example Index

Code Example 1: Working with the XcoKernel. 58
Code Example 2: Logging configuration with the TraceSource class. 75
Code Example 3: Using a service component in the traditional way. 76
Code Example 4: Using a service component in contract first design. 77
Code Example 5: Definition of a binding in the microkernel configuration. 80
Code Example 6: The kernel's standard binding configuration. 81
Code Example 7: Using the DynamicBinder to create an instance for the thread dispatcher. 82
Code Example 8: Methods for pre/post container create in the SpaceAspect class. 97
Code Example 9: Implementation of the SimpleSecurityAspect. 99
Code Example 10: Implementation of the SimpleSecurityContainerAspect. 100
Code Example 11: Using the SimpleSecurityAspect. 101

8.3 Abbreviations

API	Application Programming Interface
CCR	Concurrency and Coordination Runtime
FIFO	First In First Out
HTTP	Hypertext Transport Protocol
JMS	Java Message Service
LIFO	Last In First Out
MSMQ	Microsoft Message Queuing
NAT	Network Address Translation
RMI	Remote Method Invocation
RPC	Remote Procedure Calls
WCF	Windows Communication Foundation
XML	Extensible Markup Language
XVSM	Extensible Virtual Shared Memory

9 References

1. World Internet Usage Statistics News and Population Stats. *Internet World Stats.* [Online] Last visited: 07.09.2008. http://www.internetworldstats.com/stats.htm.

2. **Westphal, Ralf.** OOP 2008: Was kommt nach OLTP? - oder: Wo ist die Killeranwendung für Microsoft Oslo. *One Man Think Tank Gedanken.* [Online] 12 18, 2007. http://ralfw.blogspot.com/2007/12/oop-2008-was-kommt-nach-oltp-oder-wo.html.

3. *Skype.* [Online] Last visited: 07.09.2008. http://www.skype.com/.

4. **Kühn, eva, Riemer, Johannes and Joskowicz, Geri.** *XVSM (eXtensible Virtual Shared Memory) Architecture and Application.* TU-Vienna, Insititute of Computer Languages, SBC-Group : Technical Report, 2005.

5. **Gelernter, David.** Generative communication in linda. *ACM Trans. Program. Lang. Syst.* 1985.

6. **Schreiber, Christian.** *Design and Implementation of MozartSpaces, the Java Reference Implementation of XVSM: Core Structure, Transactions and Communication.* TU-Vienna, Insititute of Computer Languages, SBC-Group : Master Thesis, in preparation, 2008.

7. **Eugster, Patrick Th., et al.** The Many Faces of Publish/Subscribe. *ACM Computing Surveys.* 2003, Vol. 35, 2.

8. **Sun.** Java Message Service (JMS). *Sun Developer Network.* [Online] Last visited: 07.09.2008. http://java.sun.com/products/jms/.

9. **Microsoft.** Microsoft Message Queuing. *Microsoft Corporation.* [Online] Last visited: 07.09.2008. http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx.

10. **Sun.** Remote Method Incovation. *Sun Developer Network.* [Online] Last visited: 07.09.2008. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp.

11. **Microsoft.** .Net Remoting. *MSDN.* [Online] Last visited: 07.09.2008. http://msdn.microsoft.com/de-de/library/bb979191.aspx.

12. **Karolus, Markus.** *Design and Implementation of XcoSpaces, the .Net Reference Implementation of XVSM: Coordination, Transactions and Communication.* TU-Vienna, Insititute of Computer Languages, SBC-Group : Master Thesis, in preparation, 2008.

13. **Pröstler, Michael.** *Design and Implementation of MozartSpaces, the Java Reference Implementation of XVSM: Timeout Handling, Notifications and Aspects.* TU-Vienna, Insititute of Computer Languages, SBC-Group : Master Thesis, in preparation, 2008.

14. **Creswell, Dan.** *The Blitz Project.* [Online] Last visited: 07.09.2008. http://www.dancres.org/blitz/.

15. **Freeman, Eric, Hupfer, Susanne and Arnold, Ken.** *JavaSpaces Principles, Patterns, and Practice.* USA : Addison-Wesley Professional, 1999.

16. **GigaSpaces Technologies Inc.** *GigaSpaces.* [Online] Last visited: 07.09.2008. http://www.gigaspaces.com/.

17. **SpringSource.** *Spring Framework.* [Online] Last visited: 07.09.2008. http://www.springframework.org/.

18. **Picco, Gian Pietro, Balzarotti, Davide and Costa, Paolo.** LighTS: a lightweight, customizable tuple space supporting context-aware applications. *Proceedings of the 2005 ACM symposium on Applied computing.* 2005.

19. **Kühn, eva.** Fault-Tolerance for Communicating Multidatabase Transactions. *Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS).* 1994.

20. **Kühn, eva and Nozicka, Georg.** Post-Client/Server Coordination Tools. *Coordination Technology for Collaborative Applications.* 1998.

21. *Jini.* [Online] Last visited: 07.09.2008. http://www.jini.org.

22. **IBM.** TSpaces. *IBM Research.* [Online] Last visited: 07.09.2008. http://www.almaden.ibm.com/cs/TSpaces/.

23. **Murphy, Amy L., Picco, Gian Pietro and Roman, Gruia-Catalin.** LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.* 2006, Vol. 15, 3.

24. **Cremonini, Marco, Omicini, Andrea and Zambonelli, Franco.** Coordination and Access Control in Open Distributed Agent Systems: The TuCSoN Approach. *COORDINATION 2000, LNCS 1906.* 2000.

25. **Cabri, Giacomo, Leonardi, Letizia and Zambonelli, Franco.** Reactive Tuple Spaces for Mobile Agent Coordination. *MA '98: Proceedings of the Second International Workshop on Mobile Agents.* 1998.

26. **Androutsellis-Theotokis, Stephanos and Spinellis, Diomidis.** A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys.* 2004, Vol. 36, 4.

27. *Hibernate.* [Online] Last visited: 07.09.2008. http://www.hibernate.org/.

28. **Gray, Jim and Reuter, Andreas.** *Distributed Transaction Processing: Concepts and Techniques.* s.l. : Morgan Kaufman, 1993.

29. **Kiczales, Gregor.** Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming.* 1997, Vol. 1241.

30. **Eide, Petter L. H.** *Quantification and Traceability of Requirements.* Norwegian University of Science and Technology : Software Engineering Depth Study, 2005.

31. **Hill, Mark D.** What is scalability? *ACM SIGARCH Computer Architecture News.* 1990, Vol. 18, 4.

32. **Welsh, Matt, Culler, David and Brewer, Eric.** SEDA: An Architecture for Well Conditioned, Scalable Internet Services. *Eighteeth Symposium on Operating Systems Principles (SOSP-18).* 2001.

33. **Voelter, Markus, et al.** Patterns for asynchronous invocations in distributed object frameworks. *Proceedings of 8th European Conference on Pattern Languages of Programs (EuroPlop 2003).* 2003.

34. **Microsoft.** .Net Framework Developer Center. *MSDN.* [Online] Last visited: 07.09.2008. http://msdn2.microsoft.com/en-us/netframework/default.aspx.

35. *XCOORDINATION Software Technologies.* [Online] Last visited: 07.09.2008. http://www.xcoordination.com/.

36. *MozartSpaces.* [Online] Last visited: 07.09.2008. http://www.mozartspaces.org.

37. **Microsoft.** Windows Communication Foundation. *MSDN.* [Online] Last visited: 07.09.2008. http://msdn2.microsoft.com/en-us/netframework/aa663324.aspx.

38. **Richter, Jeffrey.** Concurrent Affairs: Concurrency and Coordination Runtime. *MSDN.* [Online] Last visited: 07.09.2008. http://msdn2.microsoft.com/en-us/magazine/cc163556.aspx.

39. **Apache.** *Apache log4net.* [Online] Last visited: 07.09.2008. http://logging.apache.org/log4net.

40. **Westphal, Ralf.** Am Anfang war der Vertrag, Contract First Design und Microkernel-Frameworks. *dotnetpro.* 06/2005.

41. **Westphal, Ralf.** Spicken nicht erlaubt, Contract First Design und Microkernel-Frameworks. *dotnetpro.* 09/2005.

42. **Martin, Robert C.** *Agile Software Development, Principles, Patterns, and Practices.* s.l. : Prentice Hall, 2002.

43. **Sun Microsystems.** *Core J2EE Patterns - Service Locator.* [Online] Last visited: 07.09.2008. http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html.

44. **Microsoft.** Singleton. *MSDN.* [Online] Last visited: 07.09.2008. http://msdn2.microsoft.com/en-us/library/ms998426.aspx.

45. **Barnett, Michael.** ILMerge. *Microsoft Research.* [Online] Last visited: 07.09.2008. http://research.microsoft.com/~mbarnett/ILMerge.aspx.