FAKULTÄT FÜR !NFORMATIK

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

Coordination, Transactions and Communication

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Markus Karolus
Matrikelnummer 0225688

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, 06.12.2009 ______________________ (Unterschrift Verfasser) ______________________ (Unterschrift Betreuerin)

Eidesstattliche Erklärung

Karolus Markus, Vertexgasse 17, 1230 Wien

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

Wien, 06.12.2009, ______________________

Abstract

We live in the era of the internet which makes it possible to connect more and more people to each other. For this large amount of users suitable technologies are needed, so users can collaborate with each other efficiently. The solution should be able to cover the complex details and problems which can occur in large systems, like concurrent access to data. The newly developed middleware XVSM (eXtensible Virtual Shared Memory) that is based on shared data structures enables an efficient solution for many real-world problems. The XVSM model allows an intuitive collaboration between different partners on a peer-to-peer infrastructure. For coordination the model supports flexible data structures that can easily be customized. Another strength of this lightweight system is that the middleware can also be used on mobile devices. The biggest advantage of XVSM is the extensibility of the module structure. New functionalities can be included very quickly to the core system.

As a basis for the descriptions in this thesis the *XcoSpaces*, the *.NET* reference implementation of XVSM, is used. The focus in this thesis lies on three main issues, which are the coordination concept, transactions and communication. First of all, many possibilities for the whole coordination mechanism are shown. The transaction and locking management allows concurrent access to data without inconsistencies. The middleware is able to communicate across machine boundaries via communication services.

Kurzfassung

Durch das Zeitalter des Internet werden immer mehr Menschen miteinander verbunden. Für die große Anzahl an Benutzern werden Technologien benötigt, die eine effiziente Zusammenarbeit zwischen allen ermöglichen. Die angestrebte Lösung sollte dabei die Komplexität, die bei großen Systemen entstehen kann, wie zum Beispiel bei gleichzeitigem Zugriff auf Daten, möglichst gut vor dem Nutzer verbergen. Eine effiziente Lösung für viele Probleme steckt in der neuen Middleware XVSM (eXtensible Virtual Shared Memory), die auf gemeinsam genutzten Datenstrukturen beruht. Das XVSM Modell erlaubt eine intuitive Kommunikation zwischen verschiedenen Parteien über eine dezentrale (P2P) Infrastruktur. Das bestehende Koordinationsmodell unterstützt flexible Datenstrukturen, die leicht angepasst werden können. Eine weitere Stärke des Systems ist die Ressourcen schonende Implementierung, die es erlaubt XVSM auch auf mobilen Geräten einzusetzen. Der größte Vorteil von XVSM liegt in der Architektur, die vor allem für Erweiterungen ausgelegt ist, durch die neue Funktionen schnell hinzugefügt werden können.

Als Grundlage für diese Diplomarbeit dient *XcoSpaces*, die *.NET* Referenz-Implementierung von XVSM. Die folgenden drei Kernthemen werden behandelt: das Koordinationskonzept, Transaktionen und Kommunikationsmöglichkeiten. Als erstes werden die sehr vielfältigen Möglichkeiten der verfügbaren Koordinationsmechanismen vorgestellt. Die Transaktions- und Sperrmechanismen erlauben gleichzeitige Zugriffe auf die Daten ohne inkonsistente Zustände. Über die diversen Kommunikationsmöglichkeiten kann die Middleware über Rechnergrenzen hinweg kommunizieren.

Acknowledgements

At this point I would like to mention several people who supported me during my studies.

Firstly I would like to thank my supervisor, *Eva Kühn* for the insightful conversations about space based computing. With her knowledge she made it possible for me to find good solutions for the given challenges in this master thesis. The helpful suggestions from *Richard Mordinyi* also inspired me during the writing process.

Ralf Westphal, an independent consultant with special skills for Microsoft Technologies like *.NET* was always very helpful with constructive ideas in the whole development process. Through the work with *Ralf* I gained a lot of experience in developing in the .NET platform.

Furthermore I want to thank *Michael Pröstler* and *Christian Schreiber* who were responsible for the *MozartSpaces* (the JAVA implementation of the XVSM space) for their exchange of ideas which helped to find solutions across technology borders.

Special thanks go to *Lucinda Monie* from Australia for helpful comments on this thesis. I would especially like to thank my girlfriend *Barbara Gerl* who endured the countless hours I have spent at the computer during my studies.

Finally, I would like to thank *Thomas Scheller* with whom I had a great time during my studies for the perfect collaboration. Together we developed the *XcoSpaces* and always found solutions even when the problems seemed unsolvable. He is a really good friend whom I can always count on.

Content

1.1 Figure Index

1.2 Example Index

1.3 Abbreviations

API	Application Programming Interface
FIFO	First In First Out
GPRS	General Packet Radio Service
GUID	Globally Unique Identifier
HTTP	Hypertext Transport Protocol
IPoint	Interception Point
LIFO	Last In First Out
MSMQ	Microsoft Message Queuing
P2P	Peer-To-Peer
RMI	Remote Method Invocation
SBC	Space Based Computing
TCP/IP	Transmission Control Protocol/Internet Protocol
WCF	Windows Communication Foundation
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XSD	XML Schema Definition
XVSM	Extensible Virtual Shared Memory
.NET	Microsoft .NET Framework

2 Introduction

The internet is continuously growing and has connected more and more people in the last years (an increase of 380 % in the past 9 years [1]). With the advent of Web 2.0 [2] the usage behavior of the internet changed drastically. At the beginning the internet provided only static content for its users. Then more and more dynamic information was offered for everyone. Nowadays modern web sites allow interacting with different people, who are now able to design and modify their own content. This trend will continue and new dynamic scenarios will be possible where users can communicate, collaborate and interact with each other.

With the growing number of users, the corresponding requests on existing systems are increasing. In traditional systems a message is sent to the server for every request and the server gives a response. When dynamic changes at the server can only be recognized over requests from the client, this behavior is also known as polling, which does not scale well if it is done too frequently. Otherwise an update could be missed if there are too few requests.

When a client needs to exchange data with another member in the network all information is routed over the server. For this purpose many different messages must be exchanged between the different parties before the actual requested information is transferred from one client to another. If many users generate concurrent requests the server can end up with a bottleneck in the system, or in the worst case, a critical server may fail and the whole service may break down.

A different approach is pursued by distributed middleware. Here the communication between the parties is decoupled. Every single participant uses a well known interface and provides its service. Instead of polling, events will be provided directly when new information has to be distributed. The required message amount will be drastically decreased, providing a good outcome for all parties involved. With middle-wares, a good solution is found for many weaknesses in the communication.

The space based paradigm goes one step further and extends the handling to the data. The data should not be stored on a centralized server. All data can be distributed between many different participants and everyone has near real-time access to the shared view of data. Every single participant can contribute its service functionality in the space. Many participants can work on the same data structures with an optimal work load [3; 4; 5]. If one worker fails, the process will be continued by the others so there is no single point of failure. The system can easily expand when the number of requests rises over a certain level.

2.1 The XVSM

The new communication paradigm is called *XVSM* [6; 7; 8] (eXtensible Virtual Shared Memory). It is based on the principles of *shared data spaces* and defines a new peer-to-peer [9] based middleware.

The idea of shared data spaces initially was born for coordinating parallel processes. These should get access to store and receive objects in virtual, associative memory. In the model, the processes have several simple operations to interact with the data structures called “tuples”. The associative memory that manages all tuples is called *tuplespace*. The whole coordination only needs four operations: inserting a new tuple, reading with remove, reading without remove and an evaluation for tuples. For the evaluation a template will be created and only the fields to be considered will be filled with data. This concept is called *Linda* and was developed by David Gelernter [10; 11].

This model has the option to obtain simple access to a large amount of data via template matching. The selection by tuples is a great method when, for example, all persons with black hair are to be selected. The system reaches its limit [7] when there is an implicit order needed for the data in the space, e.g. like in a queue.

The XVSM paradigm extends the manageability of data. For this purpose different coordination mechanisms are supported in addition to *Linda* matching. Furthermore, an elaborate event management system is included with many options for customizing the functionalities in detail. An independent communication protocol is provided for open communication across programming boundaries. To obtain access to other applications, many different transport protocols can be used together with XVSM. In addition to all supported functionalities in the space, a large effort was also given to extensibility. Using interfaces, simple access is given for new tasks; for example, a new communication service or coordination type can be added.

Currently for the XVSM model there are two implementations available. One is implemented in *JAVA* [12] and is named *Mozartspaces* [13], and the other in *.NET* [14] called *XcoSpaces* [15]. The details in this work are all based on the *XcoSpaces* implementation.

3 The XVSM Space

This chapter gives a short overview about the XVSM space. Here all functions and components will be mentioned for a better understanding of the whole system.

3.1 Introduction

For the development of the XVSM space, the experience of the *space based computing group* [16; 17] was a great benefit during the project. *Corso* [18; 19] was the first space based middleware of the Institute of Computer Languages at the Vienna University of Technology and based on the concept of *virtual shared memory* [20]. This middleware allows coordination and information exchange over shared data objects.

In the XVSM space everything revolves around containers, which manage the coordinated data structures (see Chapter 4). In containers the user data is managed and can be added and removed. The containers themselves are administered from their corresponding XVSM core instance. The core is the smallest piece of the implementation that provides all main functionalities. For more details about the core see [21].

The whole XVSM space is composed of multiple XVSM core (short XCore) instances, which are co-operating with each other. Every core instance has its own data and specialized purpose. When one instance needs data from another instance for example hosted on an external machine the information can be exchanged between them. The concepts of peer to peer networks are integrated in the communication architecture [22]. Figure 1 below shows an example of how three different core instances communicate with one another. If a container is needed from another core instance, only the address of the container is required. The access over the instance borders is automatically done by the space (for more details see Chapter 6).

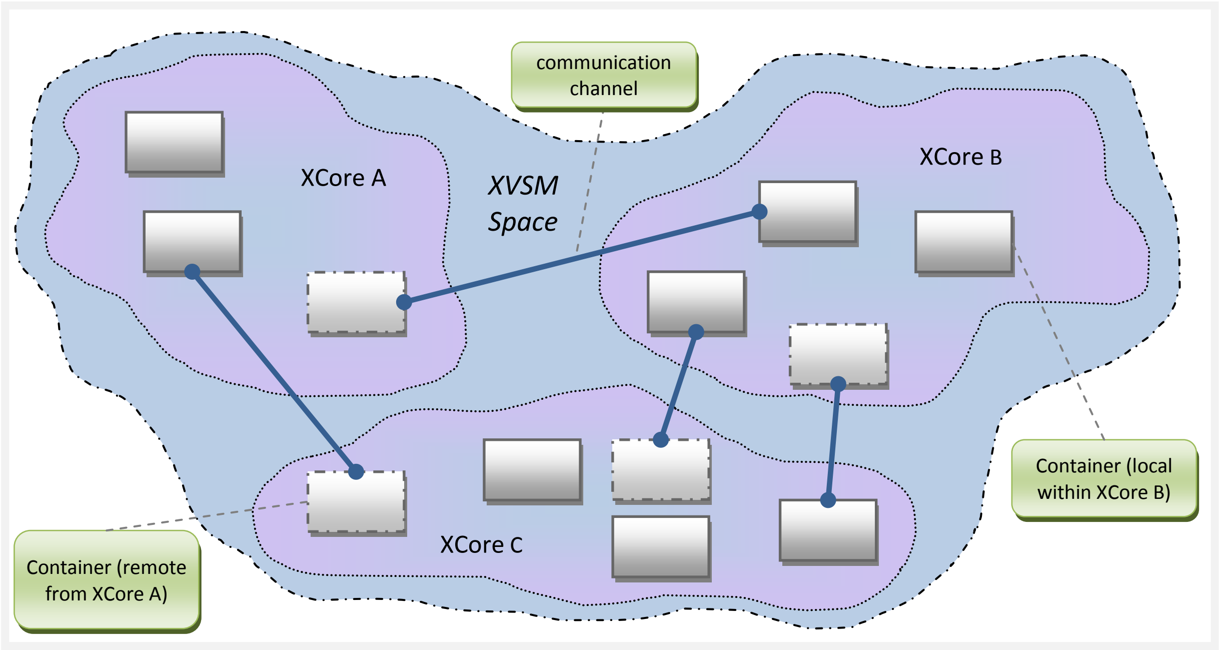

communication channel
XCore A
XVSM Space
XCore B
XCore C
Container (local within XCore B)
Container (remote from XCore A)

Figure 1: A XVSM space consisting of multiple core instances.

In the designing phase of the XVSM space we came to the decision to only implement the minimal function set for a high performing space implementation. The concrete function set can be found in [21]. All features that are not absolutely needed for the main functionality are not implemented in the core; additional functionalities could be added over profiles afterwards if needed (see [21]). The space provides different mechanisms to simply extend existing features. For example, special functionalities can be added over aspects, which can be found in Chapter 3.5.

3.2 Containers and Coordination

A very important concept in the XVSM space is the usage of containers. For accessing data within containers, only five basic operations are needed; this is similar to the *Linda* concept mentioned earlier. The basic operations are *read*, *take*, *write*, *shift* and *destroy* and more details are described in Chapter 4.4.1. The container behavior is managed by so-called coordinators. Different coordinators are allowed in a container at the same time, creating the most flexible solution. A coordinator takes care of a specialized behavior with the entries that are managed in the container. For example, the *fifo* coordinator will order the entries like a conventional queue, meaning the first entry that is inserted will be the first to be read. There is an associated selector for choosing entries for every coordinator on a container. More about selector/coordinator pairs and coordination itself can be found in Chapter 4.

The XVSM space has an additional special feature for containers. For every container, a corresponding meta container exists. The meta container information manages the details of attached container, such as creation date, maximal amount of allowed entries and much more. In addition to internal space data, user defined data can also be managed here. With meta containers, a central place for maintenance is created in the space. For more information about this see [21].

In the following table the function set is shown for different space based middleware implementations focused on coordination. Nearly all systems support the *linda* template matching. More coordination is only supported by a few solutions like *GigaSpaces* [23] or XVSM.

	Blitz [24]	GigaSpaces [23]	JavaSpaces [25]	Corso [18; 19]	XVSM	LighTS [26]
Fifo Coordination	*	*			*	
Lifo Coordination					*	
Vector Coordination					*	
Key Coordination		*			*	
Label Coordination		*			*	
Linda Coordination	*	*	*		*	*
Other Coordination				*	*	
Extensibility					*	*

3.3 Transactions and Locking

For good performance the space system should enable concurrent access to entries on containers. To ensure there is no inconsistency during multiple simultaneous read and write operations, a transaction management system with locking is supported. This takes care of all changes that are done in a single context. No other transactions can change or have access to a certain resource until the current transaction is complete; all changes are only visible to all others when the transaction is finished successfully. Detailed descriptions of transaction and locking mechanisms can be found in Chapter 5.

3.4 Core Structure

Here all components are defined which provide all required functionalities for all parts needed for processing a message in the space over all stages. For further details see [21]. First the request is recognized over the network or the embedded API. Then the operation is processed through a core processor instance. If the operation can immediately be fulfilled the response is sent back to the destination address. Otherwise the operation goes into the waiting state. Here the core must invoke all queued operations waiting for a changing event of data. On the other side, the timeout handling and exception handling is also done by the core. When it is possible (when no side effects are expected), all operations that are processed in the core are running concurrently. The locking mechanism (described in Chapter 5.4) is responsible for concurrent access to entries on a container.

When a larger number of requests appear at one core at the same time, the performance must not be compromised, so the space is dimensioned to perform on multi threaded systems. Every instance in the processing sequence has its own threading resources and is separated in a clear way over different interfaces. If a new implementation of a part is needed, it can be simply extended or exchanged.

In contrast to a huge desktop/server system, the space should also work on a small mobile device. The current implementation utilizes the existing resources carefully and is able to work on mobile devices with very few resources.

For more details about the architecture of the XVSM space see [21].

3.5 Aspects and Notifications

For better usability and extensibility, the space supports so-called aspects [27; 28]. On all important positions (see [21]) in the space the user can register to be notified for events, such as when a new container is created. These event points that are distributed are also known as interception points. For every operation a pre and a post aspect is available. The user can modify an action before it will be executed in a pre aspect. For example in a security aspect the credentials of the user can be validated before the operation is executed. In a post aspect the user receives all changes which were made and can decide to except or revoke them. The aspects themselves are divided into two groups, the first being the *space* aspects. Here the user can access all global events such as container create, transaction *create*, *commit* and so on. On the other side there are the *container* aspects, with which every access on a container can be traced. With the aspect concept the space can easily be customized. For every single event it can be decided if the event should be continued or blocked;

perhaps due to security reasons. An additional feature is that over the aspect events the space provides to change the entire behavior of the space.

Additional features such as a security mechanism can be added very easily. Over a *space* aspect, a global security instance can check every modification to containers (create, delete). With the *create container aspect* (see [21]), the security mechanism can not only monitor access; the system is also able to add its own *container* aspect to every new container, meaning the system can monitor all access to every single entry. Afterwards, the security mechanism can check every task and only needs cancel those actions that are not authorized by the aspect event.

Another popular functionality that can be implemented over aspects is persistency. Here the user can decide on which containers a persistency aspect should do its work or if the whole space should be persisted. The persistency aspect only needs to process all given changes which are resulting from *commit* transaction events.

With the aspect mechanism the available notification system is implemented. Instead of polling the status changes on a container, a notification can be registered on the container. Whenever new information is available it is automatically transferred to the user, reducing traffic for status updates to a minimum.

In the following table the function set is shown for different space based middleware implementations. All systems support notifications except *LightTS* [26] which only offers the possibilities to extend its functionality.

	Blitz [24]	GigaSpaces [23]	JavaSpaces [25]	Corso [18; 19]	XVSM	LighTS [26]
Notifications	*	*	*	*	*	
Aspects					*	
Transactions	*	*	*	*	*	

3.6 Communication

An important feature in the space is the replaceable and configurable communication service system. For best support for different communication methods, an interface is designed to allow an easy implementation of new communication modules similar to the existing ones. In the current version a *TCP socket, WCF* and *BizTalk service* implementation are supported. More information about the communication system in the XVSM space can be found in Chapter 6.

For communication a lot of the space based middleware implementations are based on JAVA. In Java the *Jini* [29] technology assists to construct distributes systems that are scalable, evolvable and flexible. This technology is used by the most of the JAVA based space implementations.

	Blitz [24]	GigaSpaces [23]	JavaSpaces [25]	Corso [18; 19]	XVSM - XcoSpaces	LighTS [26]
Jini	*	*	*			
RMI		*				
Socket				*	*	
BizTalk					*	

3.7 API

A feature that should not be left out is the application programming interface (API). This is the main connection factor to a developer using the system. Here the developer can access all methods to exploit existing features in the XVSM space. More details can be found in Chapter 7.

3.8 Other Space based Computing Systems

Since the first idea of spaces in the 1980's with the *linda* tuple space a lot of space based computing middleware solutions have been implemented. In this document some alternative middleware solutions are compared with the main focus to coordination, transactions and communication. A detailed comparison of *JavaSpaces* [25] and *Corso* can be found in [30]. *Blitz* [24], *LighTS* [26] and *GigaSpaces* [23] are compared in [21].

4 Coordination

4.1 Introduction

In the XVSM space, everything is based on coordinated data structures: so-called *containers*. Everything must be structured into containers, thus it is not possible to store data directly into the space. This structure allows a very flexible coordination. The coordination mechanism enables a hierarchical data structure to be created if needed (container references can be stored in a container). Furthermore, the container supports transactions and must deal with concurrent operations. For a detailed explanation about container functionalities in the XVSM core, see [21].

The main functionality of a container is to provide the co-ordination mechanism for *entries*. The different co-ordination types are represented through special *selectors* and *coordinators*. A *coordinator* defines how the entries must be structured for efficient coordination. For example, the coordinator decides which entries have to be read next during a *read* operation. With the *selector* the access to the entries can be specified. The selection properties can be set in the selector coordination information. A container has additional properties such as *size* and *uniqueness*. The *size* property allows a maximum count of entries in the container to be set. The *unique* property requires that only unique entries are to be stored.

Coordination functionalities can be accessed through the basic operations *read*, *write*, *take*, *shift* and *destroy*.

The read operation types:

All read operations will be blocked until the operation can be fulfilled, more information can be found in Chapter 4.5.

read*:* The entries read from the container are returned to the user. No changes on the container are made.

take*:* The concerned entries are read and removed from the container. The read entries are returned to the user.

destroy*:* The operation works like *take*, but doesn't return the read entries. Only an operation result, successful or not, is returned. This behavior spares the communication channels so not needed data won't be transferred. Because the behavior is very similar to the *take* operation, *destroy* is defined as a *read* operation.

The write operation types:

write: The entries will be added to the container. If the container is full or, for example, the key entry already exists, the *write* operation is blocked until the operation can be fulfilled.

shift: This write operation never blocks. When no blocking problem occurs the *shift* operation works like a normal *write*. Otherwise the *shift* operation removes entries according to the container's coordination types as long as it is needed to write the new entries successfully into the container.

It is important for all these operations to run successfully that the results are consistent. This is guaranteed through a fine grained locking mechanism and transactions (see Chapter 5). The support of transactions ensures that more than a single user is able to use the space at the same time and no unexpected results will be generated. The simplest locking mechanism is *container* locking. Here the whole container will be locked during a transaction. The *entry level* locking mechanism is more complex. Here the transaction only locks the required entries, so many different transactions may be active at the same time on a single container. When these transactions do not target the same entries, this locking mode will have better performance than container locking because the transactions can run in parallel (and not sequentially).

One of the most important features of the XVSM space is the support of different coordination types. The main question is how data should be stored and loaded from the space. One very widely spread coordination type in different space implementations is *Linda* [10; 11] (see Chapter 4.3.7). *Linda* is the basic concept of the *Linda* tuple space model. In this model, every data object is stored in tuple representation. In order to read the data from the space, a matching tuple template must be created. With this type of coordination unsorted data can be stored very quickly into the space because the data is independent and so comparatively easy for concurrent processing.

For some problems *Linda* is not the most suitable solution, because the data as a whole is completely unordered. In some situations a certain order is important; in this case better solutions for coordination than *Linda* are available. For example, to build up a queue in a shop application, a better solution would be an implicit order like *fifo* (first in first out).

The current implementation of the XVSM space supports the coordination types *fifo, lifo, key, list, vector, label* and *linda*. If more than these standard coordination types are required for a particular situation, the space provides an extensible interface for implementing your own coordination type, as is shown in Chapter 4.7.

4.2 Coordination mechanism

4.2.1 Coordinators

A coordinator must support all basic operations (*read*, *take*, *destroy*, *write*, *shift*) and can administrate necessary coordination information on a container. Additionally the coordinator provides a method for reading properties (metadata of the coordinator itself; more details about the meta container can be found in [21]). The information would also be manageable in the container's *meta container* but this would result in a performance decrease of the coordinator implementation. Due to concurrency reasons the coordinator must provide full support for transactions. Every single coordinator has the ability to define its transaction details in a transaction log (see Chapter 5.3) for a successful *commit* or *rollback* action. Beyond that, every coordinator can specify the locking granularity. The space supports *entry* or *container* level locking as mentioned above. For flexibility multiple coordinators can be added to a container.

4.2.2 Selectors

With a *selector* a query can be specified for an according coordinator. A selector allows specifying how many entries shall be returned. When the given selected count is -1 all matching entries are returned. Beside the count functionality, query information can be provided for the corresponding coordinator. The combination of related selector and coordinator is called selector and coordinator pair. For example the *KeySelector* and *KeyCoordinator* provide the *key* coordination functionality for the space. For the query, the key for selection can be specified in the *KeySelector*.

4.2.3 Selector & Coordinator Pairs

The processing of a single selector and coordinator pair during a read operation works as follows: First the defined selector in the *read* operation will be mapped to the coordinator instance on the container. Then the selector will ask the associated coordinator on the container which entries should be read. All read entries are given back in one result list.

In the write operation, beside the entries that shall be written, selector information can be added for every entry. The corresponding coordinator to the given selector on the container will be selected. With the selector information the coordinator can decide how to process the given entry.

4.2.4 XVSM Query Language

The XVSM space supports more than simple template matching as coordination mechanism. For selecting entries the XVSM Query Language [8; 31] is defined. In an XVSM query (XQ) several simple XVSM queries (SXQ) can be combined and chained via a pipe operator "|". In the implementation an XQ is represented through selectors. The processing of an SXQ is internally done by the coordinator functionality.

The query process begins with evaluating the first SXQ from a XQ over the whole container. The resulting multiset (bag) or sequence of matching entries is used as input for the second SXQ in the chain and so on which is shown in Figure 2.

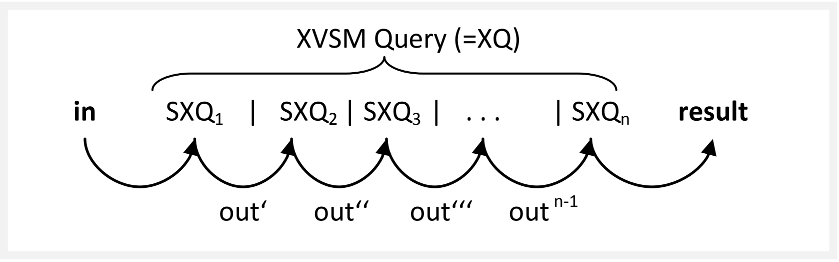


Figure 2: Execution semantic of an XQ [8]

In the implementation for a SXQ in the XcoSpaces the selector allows to specify how many entries shall be read. Additionally every selector implementation can have additional information that can be used by the coordinators. With this behavior extensible coordination models can be built with self-designed selectors that can be chained to complex queries.

For the following chapters the notation of entries, multisets and sequences are based on the *Data Model for Space-Based Collaboration Protocol* [31]. Single entries and multisets are denoted by square brackets and sequences are denoted by angle brackets. In this thesis a simplified version is used – the entry type information (e.g. *int, string*,…) is left out and all entry values are specified as anonymous labels (e.g. [type:*string*,val:"a"] → ["a"]).

Some examples:

Single entry:	["a"]
Multiset (e.g. *key container*):	[["a"], ["b"], ["c"]]
Sequence (e.g. *fifo*, *lifo container*):	<["a"], ["b"], ["c"]>

4.2.5 General coordinator rules

The following rules are valid in general for all coordinators:

- If multiple selectors are used, an element count is only valid for the first selector. If there are not enough elements for subsequent selectors, a fault will occur
- Multiple selectors are applied in the same order as they are given (as explained in the previous chapter)
- The number of selectors that can be used is not restricted
- If a count is defined in the read operation and there are not enough entries present in the container, the operation will wait until the required number of entries is available.

4.3 Semantics

4.3.1 FIFO

Fifo (first in first out) is one of the coordination types. It works like a queue, which means the order in which the entries are read out of the container is the same as they were written into the container. The *fifo* order is implicit, which means that the user cannot decide which position a new entry can be written into, or which entry can be read next. The *fifo* order can be used for many different situations, for example a printer spooler, to print the documents in the order they were sent to the printer.

4.3.1.1 Rules

- The elements are read in the order in which they were written into the container, beginning with the one that has been written first.
- When an element is written into a full container with *shift* operation, the element which would have been taken from the container next in *fifo* order is removed.
- Locking granularity for the *fifo* coordinator is container locking, for more details see Chapter 5.4

4.3.1.2 Illustrations

The following illustrations (4.3.1.2.x) are based on an unbounded container (except for *shift* where the container is bounded) with *fifo* coordination as shown in Figure 3. The entries in the container were inserted in the order <[“a”], [“b”], [“c”], [“d”]>. *Fifo* has a pointer to the first and the last inserted entry.

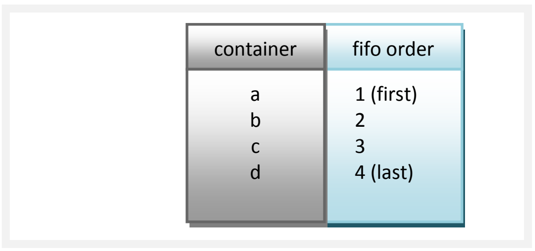


Figure 3: Example, of a *fifo* coordinated container

4.3.1.2.1 Writing an Entry

With the *write* operation a new entry will be added behind the last one. When adding an entry, for example [“e”], it will be set at position 5 and will so become the new last entry. This is shown in Figure 4.

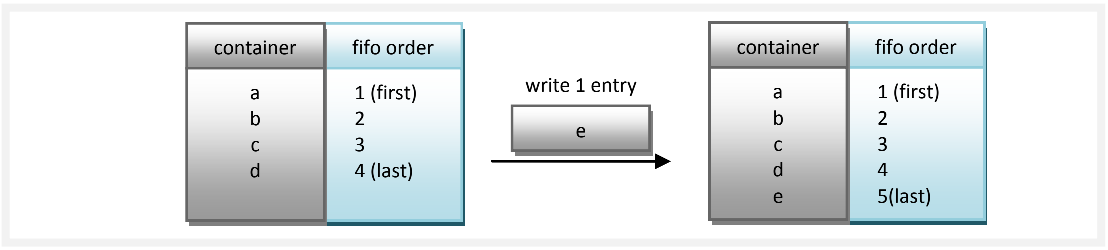


Figure 4: ***Writing*** **an entry into a** ***fifo*** **coordinated container**

4.3.1.2.2 Shifting an Entry

The following illustration (Figure 5) shows a *shift* operation on a container that is bounded to a maximum size of 4. The boundary of the container is shown in the left upper corner and with a red border. The container is full, so the *shift* operation must remove an entry before [“e”] can be written. In a *fifo* coordinated container, the entry to be read next is removed, in this case [“a”] (the one at the first position). After this entry is removed, the new entry can be written into the container and becomes the new last entry.

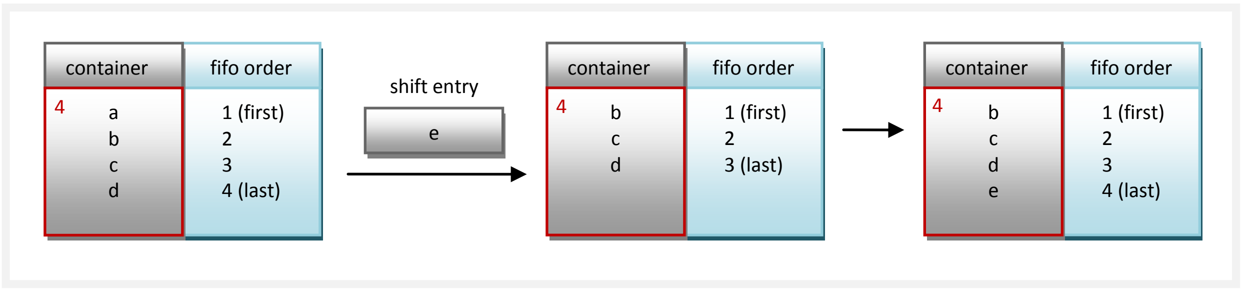


Figure 5: ***Shifting*** **an entry into a** ***fifo*** **coordinated container**

When the container is unbounded, the *shift* operation works like a normal *write*.

4.3.1.2.3 Reading an Entry

Reading an entry always means that the first entry [“a”] is read and the container is not changed. Figure 6 shows the processing of reading an entry with *fifo* coordination. The read entry [“a”] is then given back to the user and the container is not changed.

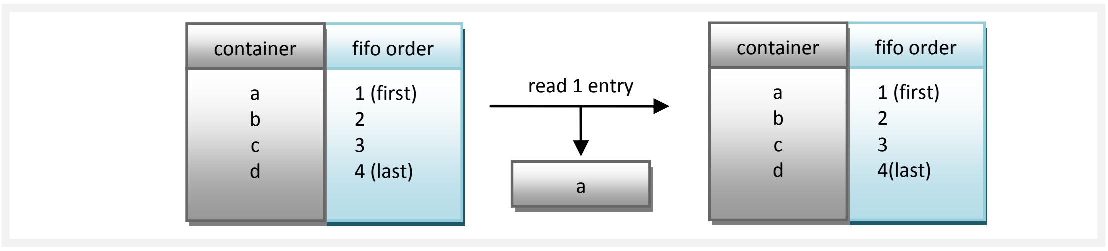


Figure 6: ***Reading*** **from a** ***fifo*** **coordinated container**

4.3.1.2.4 Taking an Entry

When an entry is taken, the first entry [“a”] is read and removed from the container, and the next entry [“b”] becomes the new first one. Figure 7 illustrates the processing of the *take* operation. The read entry [“a”] is then given back to the user.

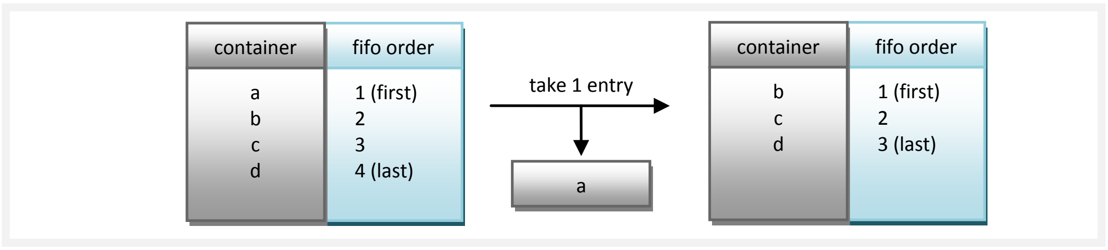


Figure 7: *Taking* an entry from a *fifo* coordinated container

4.3.1.2.5 Destroying an Entry

The operation *destroy* works like the *take* operation with the simple difference that the *destroy* operation doesn't bring the read entries back to the user. Only an operation result (successful/not successful) is given back from the *destroy* action. Figure 8 illustrates one single *destroy* for one entry. Entry ["a"] will be removed from the container.

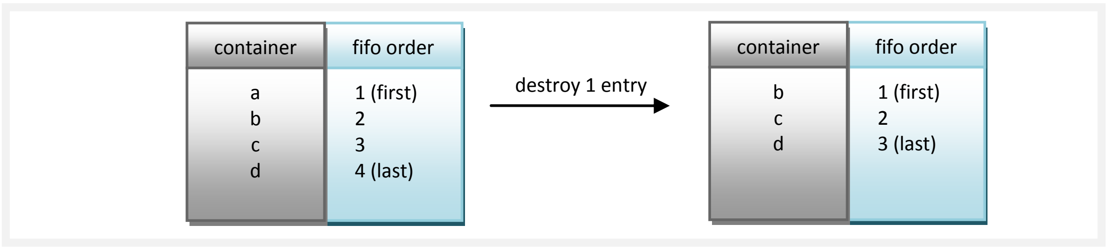


Figure 8: *Destroying* an entry in a *fifo* coordinated container

The *destroy* operation depends on the functionality of the *take* operation for all coordination types. The entries are removed over the *take* operation and the entries are never given back. In the next coordination types the *destroy* operation is not mentioned anymore.

4.3.2 LIFO

Lifo (last in first out) is another coordination type. It equals a stack, which means that the last entry that has been written into the container will be read first. The *lifo* order is also implicit like the *fifo*. So you cannot influence in which position a new entry can be written or be read next.

4.3.2.1 Rules

- The elements are read in the order in which they were written into the container, beginning with the one that has been written last.
- When an element is written to a full container with *shift* operation, the element that would have been taken from the container next in *lifo* order is removed.
- Locking granularity for the *lifo* coordinator is container locking, for more details see Chapter 5.4

4.3.2.2 Illustrations

The following illustrations (4.3.2.2.x) are based on an unbounded container (except for *shift* where the container is bounded) with *lifo* coordination as shown in Figure 9. The entries in the container were inserted in the order <["a"], ["b"], ["c"], ["d"]>. *Lifo* holds a pointer to the last inserted entry.

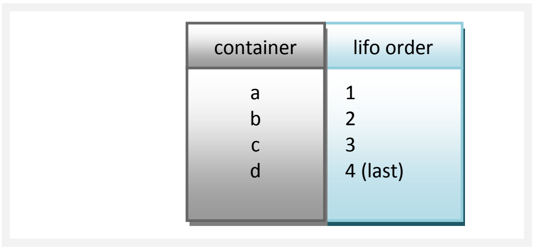


Figure 9: Example, of a *lifo* coordinated container

4.3.2.2.1 Writing an Entry

When an entry is written into the container, it is inserted behind the last entry in the container [“d”], and becomes the new last entry. This is shown in Figure 10.

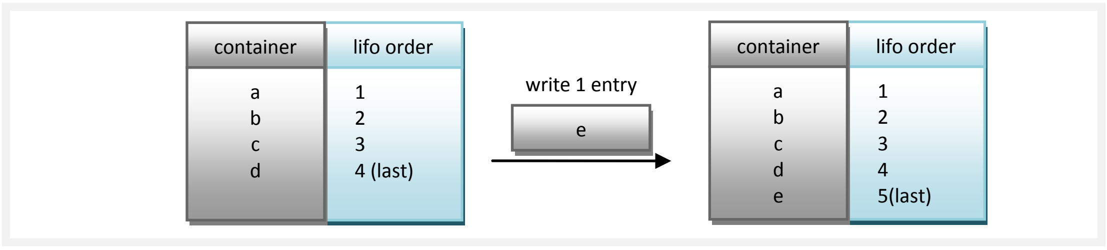


Figure 10: *Writing* an entry into a *lifo* coordinated container

4.3.2.2.2 Shifting an Entry

The following illustration (Figure 11) shows a *shift* operation on a container that is bounded to a maximum size of 4. The container is full, so the *shift* operation must remove an entry before [“e”] can be added. In a *lifo* coordinated container, the entry which would be read next is removed, which is the one at the last position (in the example this would be [“d”]). After the entry is removed, the new entry is written into the container and becomes the new last entry.

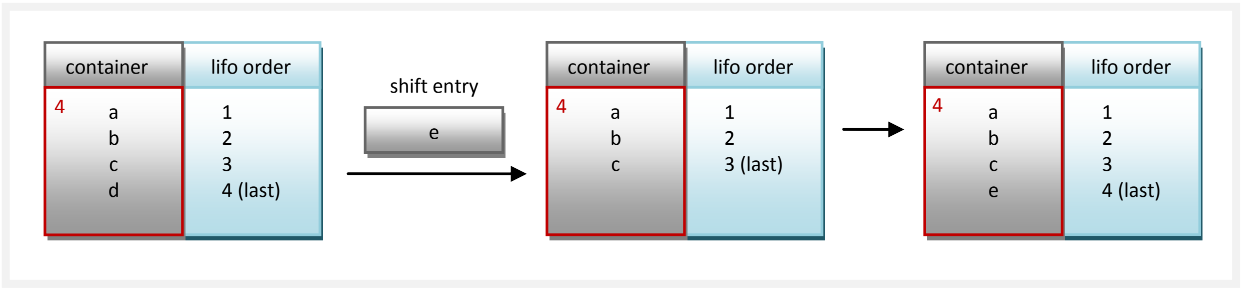


Figure 11: *Shifting* an entry into a *lifo* coordinated container

When the container is unbounded, the *shift* operation works like a normal *write*.

4.3.2.2.3 Reading and taking an Entry

The *read* operation reads the last entry [“d”] and gives back the value to the user.

When an entry is taken, the last entry [“d”] is read and removed from the container and the entry before [“c”] becomes the new last one. Figure 12 illustrates the processing of the *take* operation. The read entry [“d”] is then given back to the user.

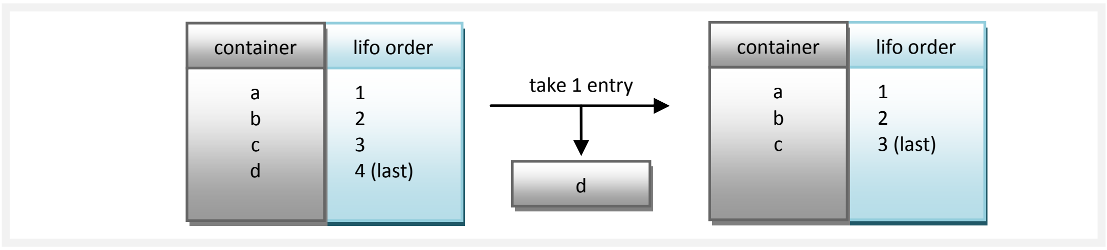


Figure 12: *Taking* an entry from a *lifo* coordinated container

4.3.3 Vector

The next coordination type is *vector*. This type acts like a linked list and is named after *JAVA*´s Vector class. Entries can be accessed by their index, beginning with zero for the first entry, and ending with (number of entries minus 1) for the last entry. Here the explicit order is combined with an implicit order which means that the index automatically changes when other entries are inserted at or removed from a lower index, so this coordinator is a hybrid. A *vector* is defined by a name on a container, so you can add different *vector* coordinators to one container.

4.3.3.1 Rules

- General *vector* index behavior
  - If an element is removed from the container with a *take* or *destroy* operation, the index of all elements that have an index greater than the removed element is reduced by 1.
  - When an element is written to the container, the index of all elements greater or equal to the index where the new element is inserted is increased by 1.
  - If "LAST" is given as an index (-1) for a new entry (or no index is defined), it will be inserted after the last entry (the one with the highest index).
  - If the index to insert an entry is greater than the number of elements in the container, a fault will be thrown.
  - If the index for a read operation is greater than the number of elements in the container, a fault will be thrown.
- Vector operations on bounded containers:
  - If an entry shall be read from a bounded container and the read index is greater than the maximum size of the container (and thus the element can never exist), a fault will be thrown.
  - If the index to insert an entry is greater than the maximum size of a bounded container, a fault will be thrown.
  - If the write operation works as *shift* and the bounded container is full, the last element (the one with the highest index) will be removed.
  - If an element is inserted at the last position with *shift* to a full container, it replaces the current last element.
- If the selector in a read or write operation uses a vector name that is not defined in the container, a fault will be thrown.
- Locking granularity for the *vector* coordinator is container locking, for more details see Chapter 5.4.

4.3.3.2 Illustrations

The following illustrations (4.3.3.2.x) are based on an unbounded container (except for *shift* where the container is bounded) with a *vector* coordination named "v" as shown in Figure 13. The entries in the container are assigned to an index by the *vector* <["a",v:0], ["c",v:1], ["b",v:2], ["d",v:3]>.

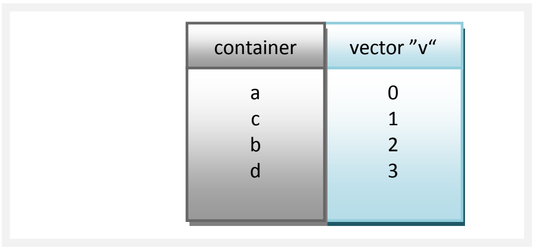


Figure 13: Example, of a *vector* coordinated container

4.3.3.2.1 Writing an entry (without a specified index)

When an entry is written into the same container as above, and this entry doesn't define a selector with a *vector* index, it is always inserted at the end of the *vector* (and its index is number of entries in the container minus 1). Here entry [“e”] is inserted at index 4 (Figure 14).

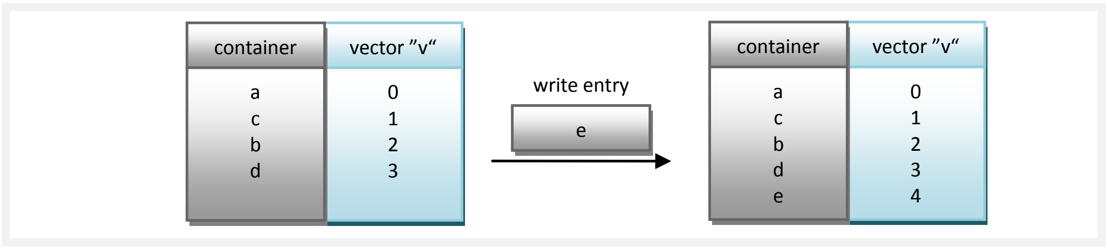


Figure 14: *Writing* an entry without a specific index into a *vector* coordinated container

4.3.3.2.2 Writing an entry to a specified index

When an entry defines a selector with a specific index for this new entry, it is inserted at this position. Entry [“f”] is inserted at index 1 (see Figure 15), the indices of entries [“b”], [“c”] and [“d”] are increased by one.

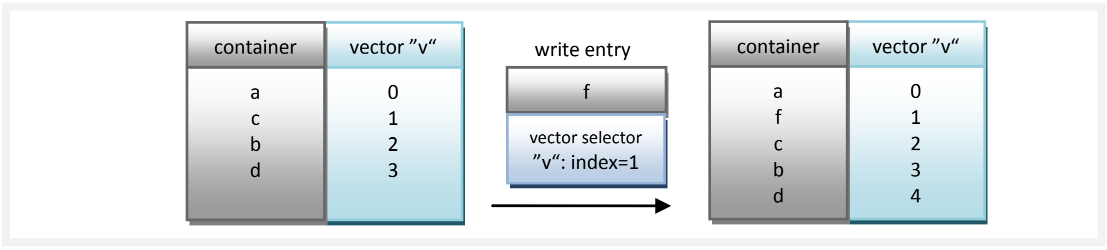


Figure 15: *Writing* an entry to a specific index into a *vector* coordinated container

4.3.3.2.3 Shifting an Entry

The following illustration (Figure 16) shows a *shift* operation on a container that is bounded to a maximum size of 4. The container is full, so the *shift* operation must remove an entry before [“e”] can be written. In a *vector*, the entry with the highest index is always removed. In this case [“d”] is removed, before [“e”] can be written to index 1.

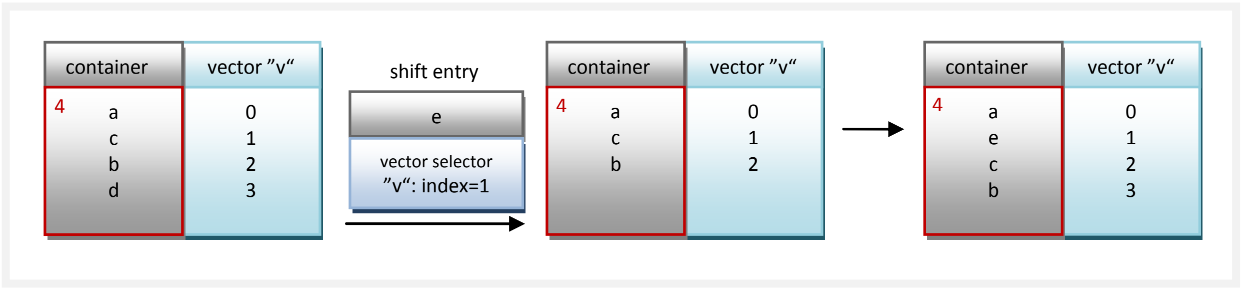


Figure 16: *Shifting* an entry into a *vector* coordinated container

When the container is unbounded, the *shift* operation works like a normal *write*.

4.3.3.2.4 Reading and taking an Entry

The *read* operation reads the entry with the given index (or indices, if more elements shall be read). For example the entry with selector index = 1 reads entry [“c”] and gives back the value to the user.

When an entry is taken, it must specify the index (or indices, if more elements shall be taken) from where it shall be read. The selector specifies index = 1, which is the entry [“c”] (see Figure 17). When [“c”] is removed, the index of [“b”] and [“d”] is decreased by 1.

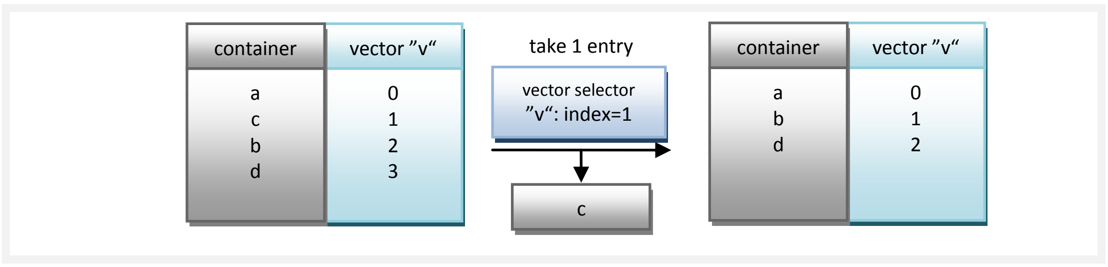


Figure 17: *Taking* from a *vector* coordinated container

4.3.4 List

The equivalent to the *Vector* implementation in *JAVA* is the *List* implementation in *.NET*. For a better support of the *.NET* programmers, we decided to implement our own list coordination type. The only difference to the behavior of a *vector* coordinator is that *shift* with an index can overwrite the entry in the container when it is specified (using the selector parameter *override*). With our implementation the behavior is more similar to *.NET’s List<>* class and we don’t lose the interoperability functionality with the *JAVA* implementation of XVSM. The list coordinator is a unique feature for the *XcoSpaces* implementation.

4.3.4.1 Rules

All rules are valid from the vector rule set (see Chapter 4.3.3.1 Rules).

- If the *write* operation is a *shift*, the element with the given index will be replaced when the override parameter is used in the selector.

4.3.4.2 Illustrations

4.3.4.2.1 Shifting an Entry

The following illustration (Figure 18) shows a *shift* operation on a container. In the *list*, the entry, with the given index, is removed when in the selector property override is set to true. In this case [“e”, index:1] should be shifted. So entry [“c”] will be removed, then [“e”] is written to index 1.

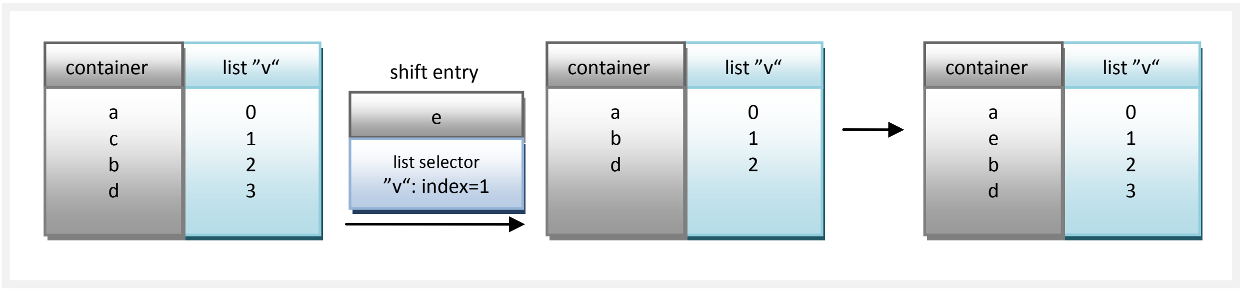


Figure 18: *Shifting* an entry into a *list* coordinated container

4.3.5 Key

The *key* coordination type acts like a hash table – the entries are coordinated by a key. This coordination type supports genericity, so you have type safe access to all keys and values. The *Dictionary*<> class in *.NET Framework* is comparable to this coordination type. This coordination type has its entries only ordered explicitly, because all coordination decisions are made on the key information. This information is created during the *write* operation and will not be changed until the data is removed from the container. Based on this behavior, this coordination type implements entry locking (see Chapter 5.4). During an operation, only a single entry will be changed, so it is comparatively easy to implement entry locking compared to other coordination types. In other coordination types like *lifo*, a single entry change will have effects to the whole container information.

One *key* coordinator on a container is defined by a name and type, so more than one instance for a container is supported. If a *key* coordinator is defined for a container, entries can be given a key value for this coordinator. Comparable to a database a key value must be unique, it can only be used once. Unlike in a DB, an entry may define no key value at all. When reading from a container, entries can be accessed by their key values.

4.3.5.1 Rules

- All keys are unique.
- Not every entry must define a key
- If the selector of an entry that shall be written to a container uses a key name that is not defined in the container, or if the type of the key is different than the one defined, a fault will be thrown.
- If the key of an entry that shall be written to a container is already present, and *write* is used as an operation, the operation blocks until the entry with this key value is taken from the container.
- If the key of an entry that shall be written to a container is already present, and *shift* is used as operation, the entry with this key value is removed from the container. This also means that if more than one key selector is defined for this entry, the operation might remove more than one entry from the container before the new entry can be written.
- When an entry must be shifted out of a bounded and full container with key coordination (and no conflicting keys exists), it is not clearly defined which entry will be chosen. The selection of the entry will be done at random.
- Locking granularity for the *key* coordinator is entry locking, for more details see Chapter 5.4.

4.3.5.2 Illustrations

The following illustrations (4.3.5.2.x) are based on an unbounded container (except for *shift* where the container is bounded) with a *key* coordination named “k” and the type is string as is shown in Figure 19. The entries in the container are ordered in the same way as a dictionary [[”a”, k:”k1”], [“b”, k:”k6”], [“c”, k:”k3”], [“d”, k:”k9”]].

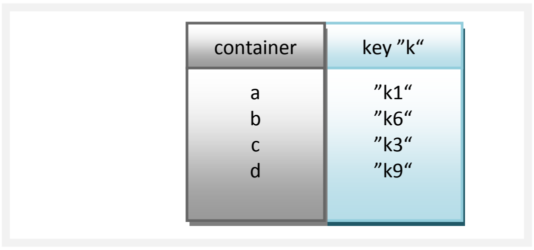


Figure 19: Example, of a *key* coordinated container

4.3.5.2.1 Writing an entry

When an entry shall be written to the container with a key that is not yet specified, it is simply added to the container without any effects to the other entries. The entry [“e”] is added to the container (Figure 20) with the key value “k4”.

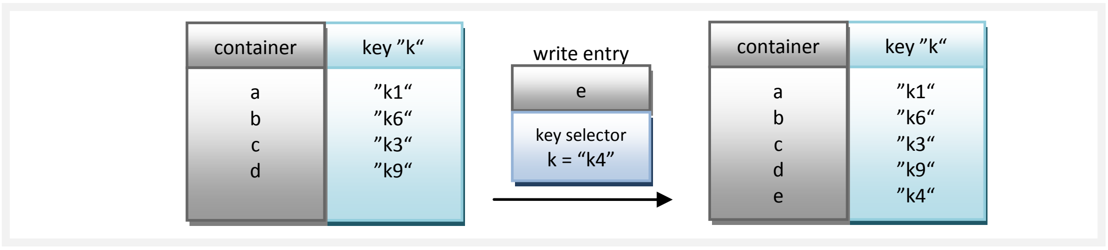


Figure 20: *Writing* an entry into a *key* coordinated container

4.3.5.2.2 Shifting an Entry

When an entry is written to a container using *shift* operation with a key value that already exists in the container, the entry with this value is removed first from the container. The new entry [“e”] has the same key value “k6” as the entry [“b”] in the container (shown in Figure 21), so [“b”] is removed from the container, and then [“e”] is inserted.

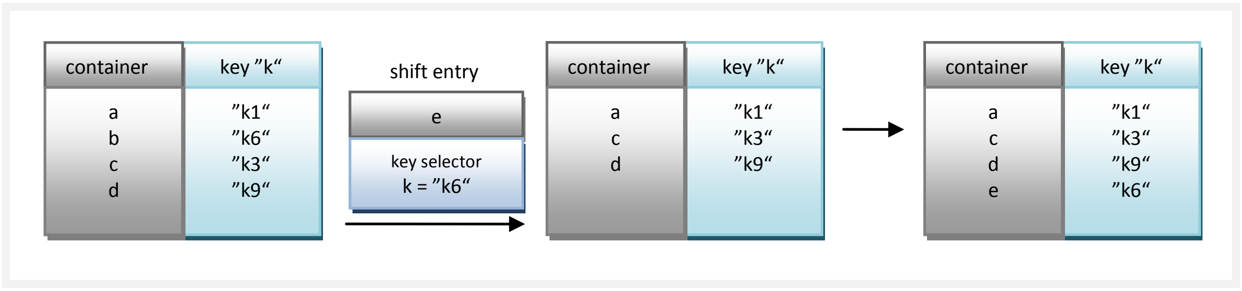


Figure 21: *Shifting* an entry into a *key* coordinated container

4.3.5.2.3 Shifting an entry to a bounded and full container

When an entry is written to a bounded and full container using *shift* operation with a key value that doesn’t already exist in the container, an entry must be removed. This will be chosen at random. In the example e.g. entry [“a”] (in Figure 22) is removed first and then [“e”, k:”x1”] is written.

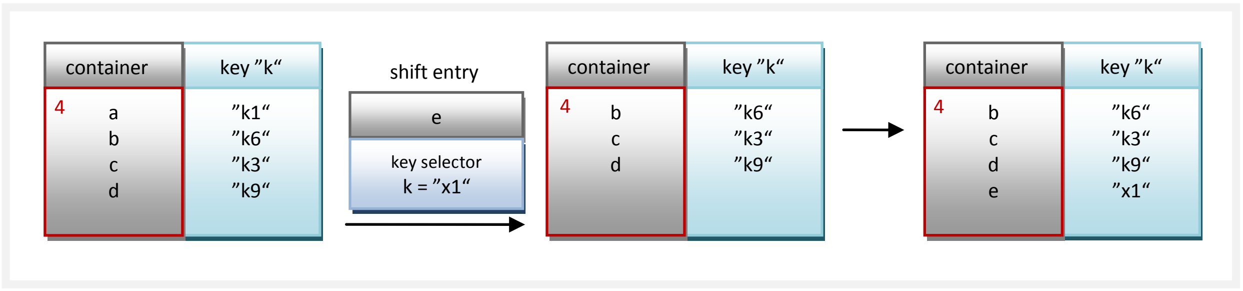


Figure 22: *Shifting* an entry into a full *key* coordinated container

When the container is unbounded, the shift operation works like a normal *write*.

4.3.5.2.4 Reading and taking an Entry

The *read* operation reads the entry with the given key (or keys, if more entries shall be read). For example, an operation with selector key (k:"k3") reads entry ["c"] and gives back the value to the user.

When an entry is taken, the operation must specify the key value (or values, if more entries shall be taken) that shall be read. The key value "k6" in Figure 23 belongs to the entry ["b"], so take "k6" reads and removes ["b"] from the container.

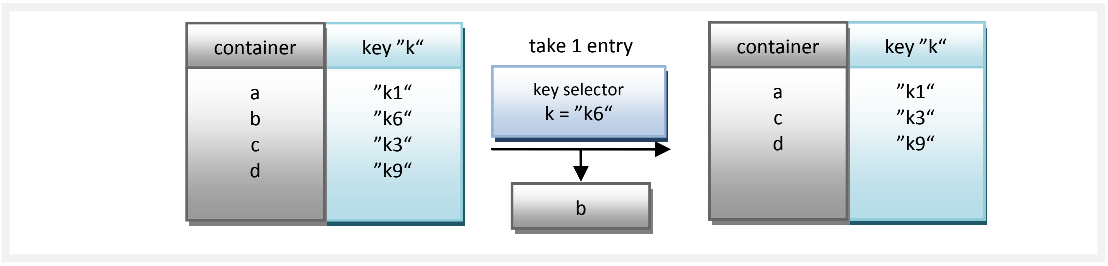


Figure 23: *Taking* an entry from a *key* coordinated container

4.3.6 Label

The *label* coordinator type is very similar to the *key* coordination type, with one very important difference; with *label* you can have many different entries with the same label name. This coordination type also supports genericity.

4.3.6.1 Rules

- Multiple entries can have the same label name.
- Not every entry must define a label
- If the selector of an entry that shall be written to a container uses a label name that is not defined in the container, or if the type of the label name is different than the one defined, a fault will be thrown.
- When an entry must be shifted out of a full and bounded container with label coordination, it is not clearly defined, which entry will be chosen. The selection of the entry will be done at random.
- Locking granularity for the *label* coordinator is container locking, for more details see Chapter 5.4. In a future version the implementation will be changed to entry locking.

4.3.6.2 *Illustrations*

4.3.6.2.1 Writing an entry

The following illustration shows a *write* operation to a *label* coordinated unbounded container (the *label* is identified with “l”). The entries in the container are inserted with multiple values for one label [[“a”, l:”l0”], [“b”, l:”l3”], [“c”, l:”l1”], [“d”, l:”l4”]]. When an entry defines a selector with a specific label for this new entry, it is inserted with this label. In the example (Figure 24), entry [“f”] is inserted with label “l3”.

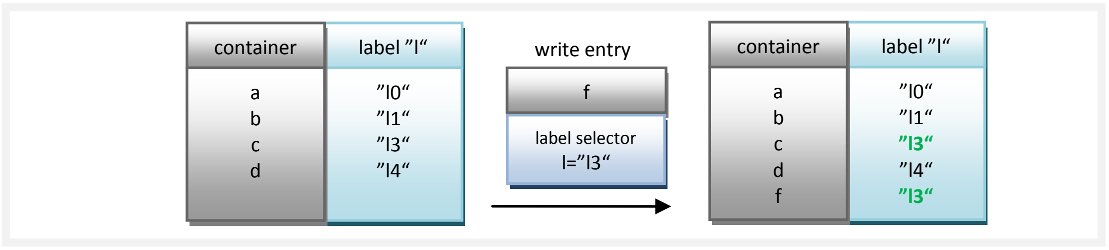


Figure 24: *Writing* an entry into a *label* coordinated container

4.3.7 Linda

The *Linda* coordinator implementation allows the user to use classical tuple space coordination. For the *Linda* model every entry must be represented in a tuple representation. A tuple represents a stored sequence of typed fields. Over this construction the template matching works, when the user wants to find the entries during a *read* operation. With this possibilities *Linda* has a simple mechanism to access single data structure parts in the container.

4.3.7.1 *Template Matching*

For the read operations Linda template matching allows different specifications for entry selection. First a complete type matching is allowed e.g. selecting all Tuples that have on the first position an integer value and on the second a string (Tuple<int, string>). The second possibility is to specify only certain required types and let the others empty. For example matching all entries with an arity of two where the one on the second position is a string (Tuple<null, string>). The third and last possibility is to define concrete values for matching e.g. all tuples where on the first position the value is an int with the value two (Tuple<2, null>).

4.3.7.2 *Rules*

- Entries have to implement the *ILindaMatchable* interface. When an entry is written that doesn’t implement this interface, a fault will be thrown.
- The *shift* operation equals a normal *write* when the container is unbounded.
- When an entry must be shifted out of a full container with *linda* coordination, it is not clearly defined which entry will be chosen. The selection of the entry will be done at random.
- Locking granularity for the *linda* coordinator is container locking, for more details see Chapter 5.4. In a future version the implementation will be changed to entry locking.

4.3.7.3 *Illustrations*

The following illustrations (4.3.7.3.x) are based on an unbounded container (except for *shift* where the container is bounded) with *linda* coordination as shown in Figure 25. The entries in the container are inserted with two different tuple types – first is a *Tuple*<int> and second a *Tuple*<int, string>. The inserted entries are [<1,"a">, <2,"b">, <3>, <4,"c">].

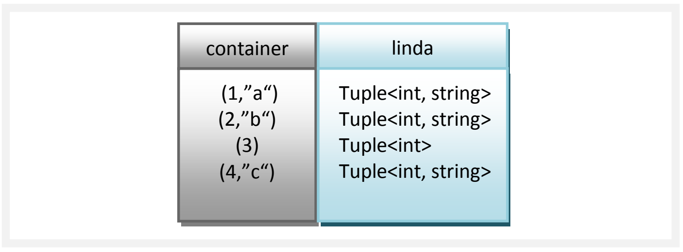


Figure 25: Example, of a *linda* coordinated container

4.3.7.3.1 Writing an entry

When an entry shall be written to the container, it is simply added to the container without any effects to the other entries. In the example the entry <5,"d"> is added to the container (in Figure 26).

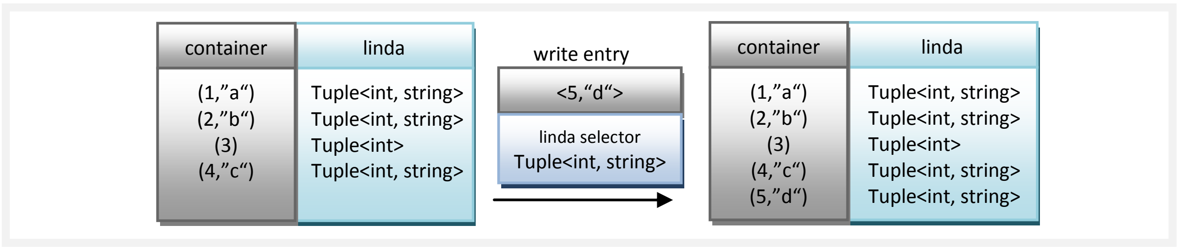


Figure 26: *Writing* an entry into a *linda* coordinated container

4.3.7.3.2 Reading and taking an Entry

The *read* operation reads all entries that are matching with the given selector template. For example a *read* operation with template *Tuple*<int, string> reads three entries [<1,"a">, <2,"b">, <4,"c">] and gives them back to the user.

With *take*, all entries matching the template are removed from the container. For example (Figure 27), the template *Tuple*<int> reads only one entry *[3]* and removes the entry from the container.

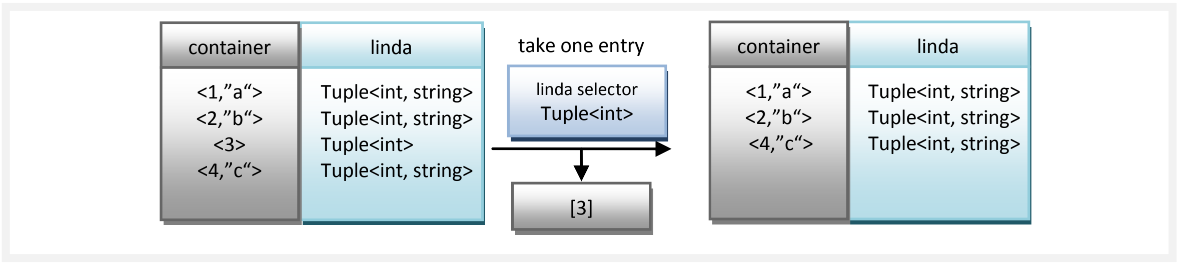


Figure 27: *Taking* an entry from a *linda* coordinated container

4.4 Implementation

The implementation of containers and concrete coordination types is strictly separated into interfaces. In this way many different coordination types can be combined on one single container. The processing is done in the order in which the coordinators are created on a container. The functionality of every coordination type is split into a *coordinator* and *selector pair*. In the *selector contract (XcoSpaces.Kernel.Contracts.Selectors)* all interfaces and classes that are needed for implementation are specified. The most important parts are the abstract class *Selector* which all *selectors* inherit from and the interface *ICoordinator* for all *coordinators*. Both definitions can be found under the namespace *XcoSpaces.Kernel.Selector* in the contract package.

For a simple identification of the selector/coordinator pairs, the names are a combination of the coordination type and the pair-type (*Selector/Coordinator).* E.g. naming convention used with *fifo: FifoSelector* and *FifoCoordinator.*

The following chapters show the concrete implementation of the different coordination types. For more detailed description of the *selector* contract, see [21].

4.4.1 Basic Operations

The next section illustrates a user code example that demonstrates the usage of all basic operations based on a container. For this sample, a container with *key* coordination is used so a simple coordination flow can be shown with generics. More details about the used commands can be found in Chapter 8. All operations in this example use implicit transactions (the parameter transaction=null) and timeout in milliseconds. For more information about transactions see Chapter 5. First a new space instance is initiated and a new local container (no remote address needed, so the first parameter is null) will be created with the generic *KeySelector<string>* with the key name “k” without timeout (second parameter the constant *System.Threading.Timeout.Infinite* for infinite). Every next part of this example depends on the previous parts.

```
//create a new space by instantiating the kernel
using (XcoKernel kernel = new XcoKernel())
{   //create a container with key coordination
    ContainerReference cref = kernel.CreateContainer(null, Timeout.Infinite,
       new KeySelector<string>("k"));
…
```

Example 1: Initialize XcoKernel and create container

4.4.1.1 Write

The next part of the example writes three new entries into the empty container. The *write* operation on a container needs the following information: the container reference of the target container, transaction reference, timeout and the data that shall be written. As shown in this example, in addition to the data that shall be written into the container for every entry the coordination information for the explicit *key* coordinator is provided. With the *KeySelector* the keys for every entry can be specified. Following entries [[”a”, k:”k1”], [“b”, k:”k2”], [“c”, k:”k3”]] will be written in the example (Example 2) into the container with *KeySelector* “k”.

```
…
    //writing entries into the container with KeySelector informations
    kernel.Write(cref, null, 1000, new Entry("a",
             new KeySelector<string>("k","k1")));
    kernel.Write(cref, null, 1000, new Entry("b",
             new KeySelector<string>("k","k2")));
    kernel.Write(cref, null, 1000, new Entry("c",
             new KeySelector<string>("k","k3")));
…
```

Example 2: Writing into container

4.4.1.2 Read

The following example shows two read operations. The first operation reads the entry with the key “k1” and the second reads all (parameter constant *Selector.COUNT_ALL*) entries, i.e. the entries with the keys “k2” and “k3”. A single *read* operation needs, like the *write* operation, the container reference, transaction reference, timeout and the specification which data should be read. With the *KeySelector* the key selection can be made.

```
…
    //now read the entries from the container
    List<IEntry> result = kernel.Read(cref, null, 1000,
              new KeySelector<string>(1, "k", "k1"));
    //result contains the entry ["a"]
    result = kernel.Read(cref, null, 1000,
              new KeySelector<string>(Selector.COUNT_ALL, "k", "k2","k3"));
    //result contain the entries ["b"] and ["c"]
…
```
Example 3: Reading from container

4.4.1.3 Take

In the next part one entry will be taken from the container. The *take* operation works like the *read* operation with the only difference being that the entry will be removed afterwards. In the example, the entry [“c”] will be taken from the container with the key “k3”. After the successful completion of the operation only two entries remain in the container [[“a”, k:”k1”], [“b”, k:”k2”]].

```
…
    //now take a entry from the container
    result = kernel.Take(cref, null, 1000, new KeySelector<string>(1, "k", "k3"));
    //result contains the entry ["c"]
…
```
Example 4: Taking from container

4.4.1.4 Shift

The following example shows a *shift* operation. The entry [“b”, k:”k2”] will be overwritten through the new entry [“d”, k:”k2”] because both entries have the same key. The two entries [[“a”, k:”k1”], [“d”, k:”k2”]] remain in the container after the *shift* operation has finished.

```
…
    //overwrite the existing entry by a new one
    kernel.Shift(cref, null, 1000, new Entry("d",
              new KeySelector<string>("k", "k2")));

    //read entries with key k1 and k2
    result = kernel.Read(cref, null, 1000, new
    KeySelector<string>(Selector.COUNT_ALL, "k","k1", "k2"));
    //result contain the entries ["a"] and ["d"]
…
```
Example 5: Shifting from container

4.4.1.5 Destroy Container

Finally, the container will be removed and all resources will be cleaned up.

```
…   //destroy the container
    kernel.DestroyContainer(cref);
}
```
Example 6: Destroy container

4.4.2 Implemented coordination types

The following coordination types can be found in the package *XcoSpaces.Kernel.Selectors*:

- *FifoSelector & FifoCoordinator*
- *LifoSelector & LifoCoordinator*
- *KeySelector<TKey> & KeyCoordinator<TKey>*

- *LabelSelector<TLabel> & LabelCoordinator<TLabel>*
- *ListSelector & ListCoordinator*
- *LindaSelector & LindaCoordinator*

Both key and label coordinator implementation support a specialized functionality via the *GetProperty* method to provide simple access to the list of all keys/labels. This is faster than reading all entries with all data when only the keys/labels information is needed.

4.5 Processing the basic operations

For a better understanding of how the space manages the different coordinators, this chapter shows the exact procedures for all basic operations on a container.

For the basic operations, some helper objects do exist such as *RequestRead* and *RequestWrite*. The *RequestRead* contains the read operation type (*read*, *take*, *destroy*), the chosen selectors, container reference, transaction reference and the timeout. The corresponding object for the write operations (*write*, *shift*) is *RequestWrite*. The difference to *RequestRead* is, that in contrast to the *read* operation object, it holds the *write* operation type and the list of entries (in each entry the selectors for writing are included), which should be written instead of the query selector information.

All basic operation methods give back the completion status of the operation. Each time an operation cannot be fulfilled (locking, aspect break, no results) it will be rescheduled when the timeout is not reached.

4.5.1 Aspects in the space

Besides coordination, aspects provide a very powerful functionality in the space. With aspects the developer gets the possibility to extend the space behavior and functionality. On all important positions in the space there are extension markers, called *interception points* (short *IPoints*), which allow access to observe or modify functionalities in the space. This can be seen as an implementation of the interceptor pattern [32]. For all operations (except SpaceShutdown – here only pre aspects exist) there are two *IPoints* defined, one before (Pre) and one after (Post) the operation. To maximize different possibilities of every single *IPoint*, special data are provided, such as transactions, selector collections, container references and many others. For example, the developer can write the result of successful operations into a database using Post-IPoints, or write all basic operation events into a log file using Pre-IPoints.

The following return values of aspects are possible:

Reschedule:	If an aspect sets the *AspectResult* to reschedule, an *XcoAspectRescheduleException* will be thrown. The exception forces an immediate operation termination and the operation will be prepared for reschedule. This means that current changes will be reverted and the operation will be set to the waiting state.
Skip:	If the aspect result is changed to *skip*, the operation advances directly to the post aspects. This return value makes only sense for pre aspects.
Ok:	The normal processing will be continued. i.e. the next aspect will be processed or the operation will be continued when no further pre aspect exists.
NotOK:	If the operation cannot be fulfilled throw an *XcoAspectException*. Current changes will be reverted and the whole operation fails. In the following examples no special

attention will be given to this case because this simply leads to a rollback and causes the operation to fail. It will therefore not be depicted in the following figures.

For more details about the theory of aspect usage, have a look at [33] and for more practical details in the *.NET* implementation, at [21].

4.5.2 Processing read

In the following Example 7, the signature of the *read* operation is shown. The *read* operation returns a list of all resulting entries. The parameter *op* (RequestRead) contains the selector information for the reading operation.

```
public bool Read(RequestRead op, Transaction t, out List<IEntry> entries)
```

Example 7: Signature of the read method on the container

After the start of the *read* operation that is shown in Figure 28, the first action is the processing of all *PreRead* aspects over the *AspectManager*. After aspect processing, the given selectors will be checked. It is required that at least one selector is specified and that no more selectors are used as coordinators than are available on the container. When these requirements cannot be fulfilled an *XcoContainerReadException* will be thrown and the operation will fail and will not be rescheduled.

To prevent any access violations, parts of the operations must have exclusive access to the container. These are shown as green colored items in Figure 28. The first part of the main *read* operation tries to lock the whole container if the coordinators set the granularity to container locking. For more details about locking see Chapter 5.4). When the whole container must be locked and it is not possible at that particular moment (for example when another transaction is already locking the same resources), all previous actions will be rolled back and the operation will be rescheduled for a new attempt. If the locking is successful or not required, the actual read action will begin.

The chosen selector list for the *read* operation is accessible over the *RequestRead* item. In a loop over all chosen selectors in the list, the corresponding coordinators at the container will be selected. The *read* operation is done for every coordinator. If any read entries exist, every coordinator will receive a list of previously read entries. For this list every coordinator implementation must determine if the *read* operation can be fulfilled. If one of the coordinator conditions cannot be carried out, the whole operation will be rolled back and will be prepared for retry. If the specified number of entries is found after asking all selectors, the read action will be deemed to be finished successfully.

The read entry locking will now be started if the container's locking granularity is set to entry locking. Similar to the container locking mentioned previously, when the locking does not succeeded, rollback and reschedule are initiated; otherwise the process continues. After finishing this step successfully the exclusive access to the container will be terminated.

The last step in the read process after the successful search for entries is the processing of all *PostRead* aspects. Aspect *Reschedule* results in a rollback and retry of the whole operation. If the post aspect cannot be fulfilled, an *XcoAspectException* is thrown, rollback starts and the operation fails. In all other aspect states the operation returns the entry list and finishes the *read* operation successfully.

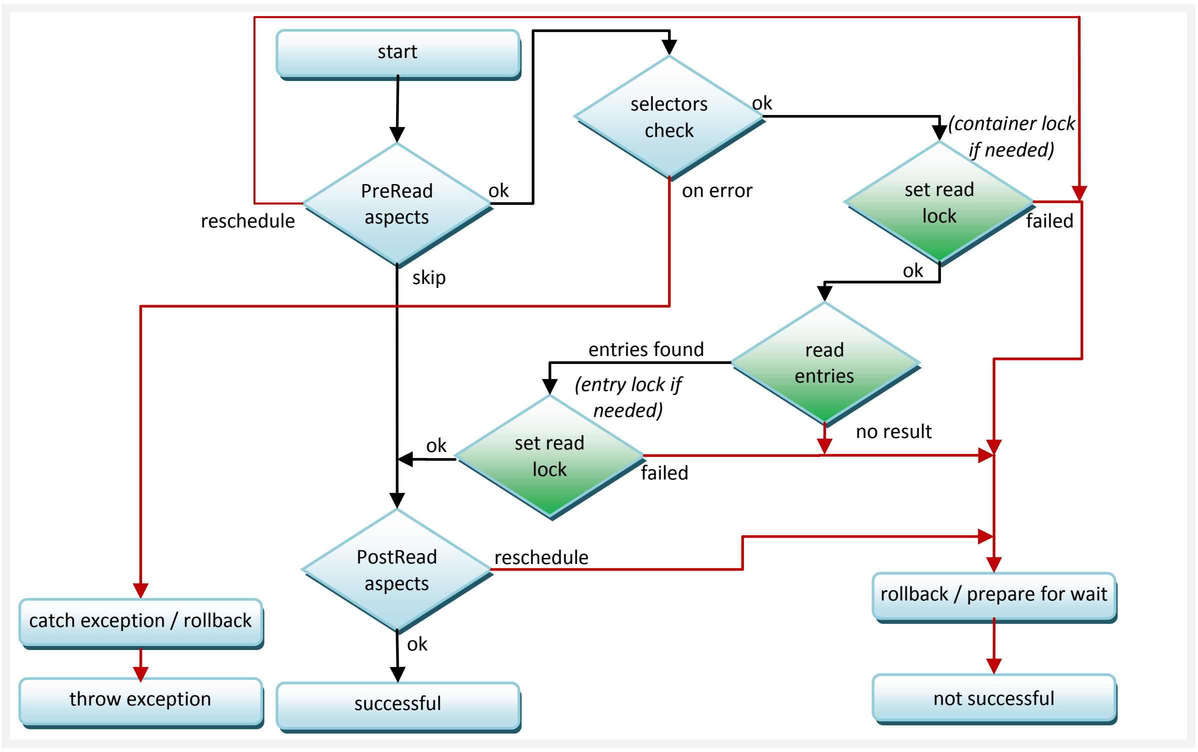


Figure 28: *Read operation on a container*

For simplicity in the following explanations for processing basic operation, the aspect processing of pre and post aspect is only hinted with gray items and dotted lines. For each basic operation an equivalent pre and post aspect operation will be performed. No special attention will be given to failed locking requests (write or read), because the behavior is always the same. The request will be cancelled, the rollback will be carried out for all changes in this operation and then the operation will be prepared for reschedule (For more details about the rescheduling system see [21]). This also means that the exclusive access will be terminated, if still existing.

As mentioned above, the next figures will only concentrate on the actions between the pre and post aspects.

4.5.3 Processing take and destroy

Take and *destroy* (shown in Figure 29) are very similar. However, there are two differences; 1) the pre and post aspect *IPoints* are different, and the 2) the *destroy* operation does not give the read entries back in contrast to *take*. The latter is reflected in the signature (Example 8) of these two methods. As mentioned before, the parameter op includes the selector information for the operation.

```
public bool Take(RequestRead op, Transaction t, out List<IEntry> entries)

public bool Destroy(RequestRead op, Transaction t)
```

Example 8: Signature of the take and destroy operation on the container

After the pre aspect processing the read selectors will be checked. This works similarly to the *read* operation. Next, the excusive processing will begin. If one of the coordinators of the container needs container locking, the following action will attempt to acquire a *write* lock for the whole container. After successful completion of the locking process, the reading procedure starts which follow the same procedure as the *read* operation (see Chapter 4.5.2). When the reading action can be successfully finished and entries have been read from the container, the *write* locking starts on entry level provided that the container is not *container* locked yet. When the locking has finished without error, all selected entries are removed from the container. The exclusive access is released and the post aspect processing starts.

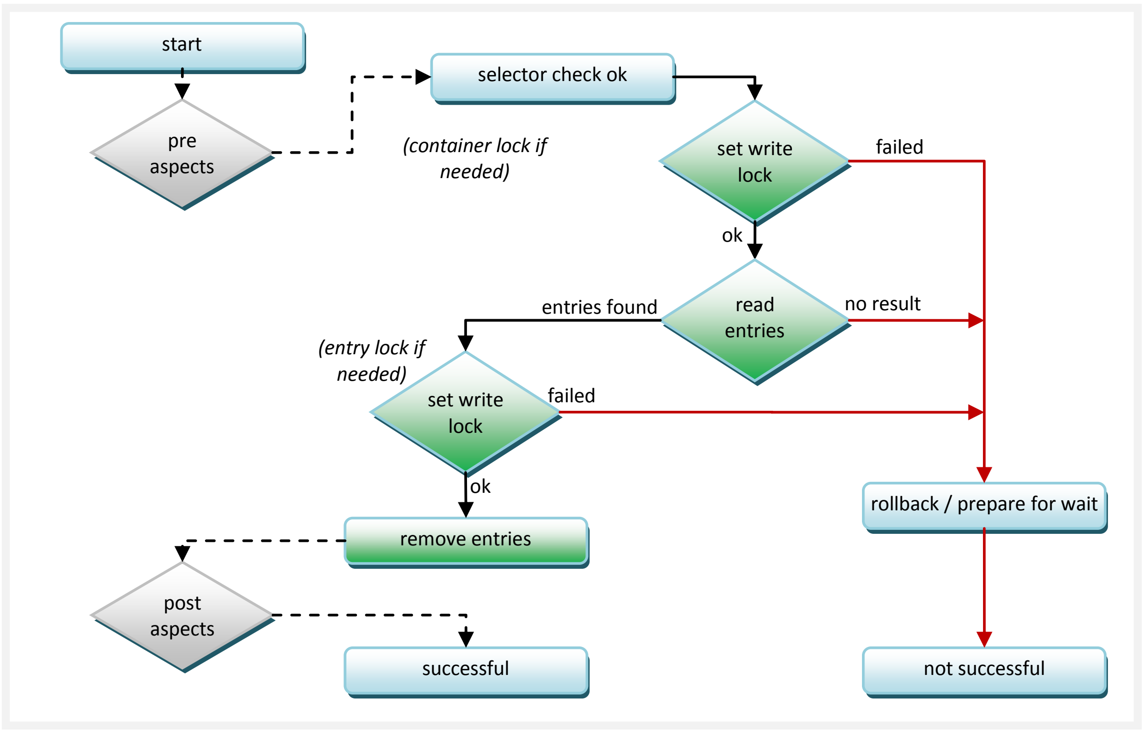


Figure 29: ***Take and destroy operation on a container***

4.5.4 Processing write

The signature for the *write* operation includes the *op* object which contains entries which need to be written including selector information and transaction.

```
public bool Write(RequestWrite op, Transaction t)
```

Example 9: Signature of the write method on the container

The first action after the pre write aspect is the start of the exclusive access (see Figure 30). The *container* write-locking starts directly after the exclusive access is granted, if the locking granularity is container locking. Then the processing starts for every single entry required to be written to the container. If the container is full, and no more entries may be added, the write operation will be canceled and rollback and rescheduling will be started. If there is enough space for a new entry in the container, the *write* entry lock will be requested, providing that entry locking is allowed and that the whole container is not already locked. After the successful locking, the write process can be started.

Every existing coordinator on the container is then allowed to create coordination data. The only response from a single coordinator is the processing result. If a coordinator cannot perform the *write* operation (e.g. because an entry with the same key already exists, in case of a key coordinator), the rollback and reschedule process are started. When all coordinators of the container have seen the entry and the coordination accounting is finished, the entry is added to the container memory. This process is repeated for every entry to be written within the given operation. If all entries can be written to the container without error, the exclusive access will be terminated and the post aspects will start.

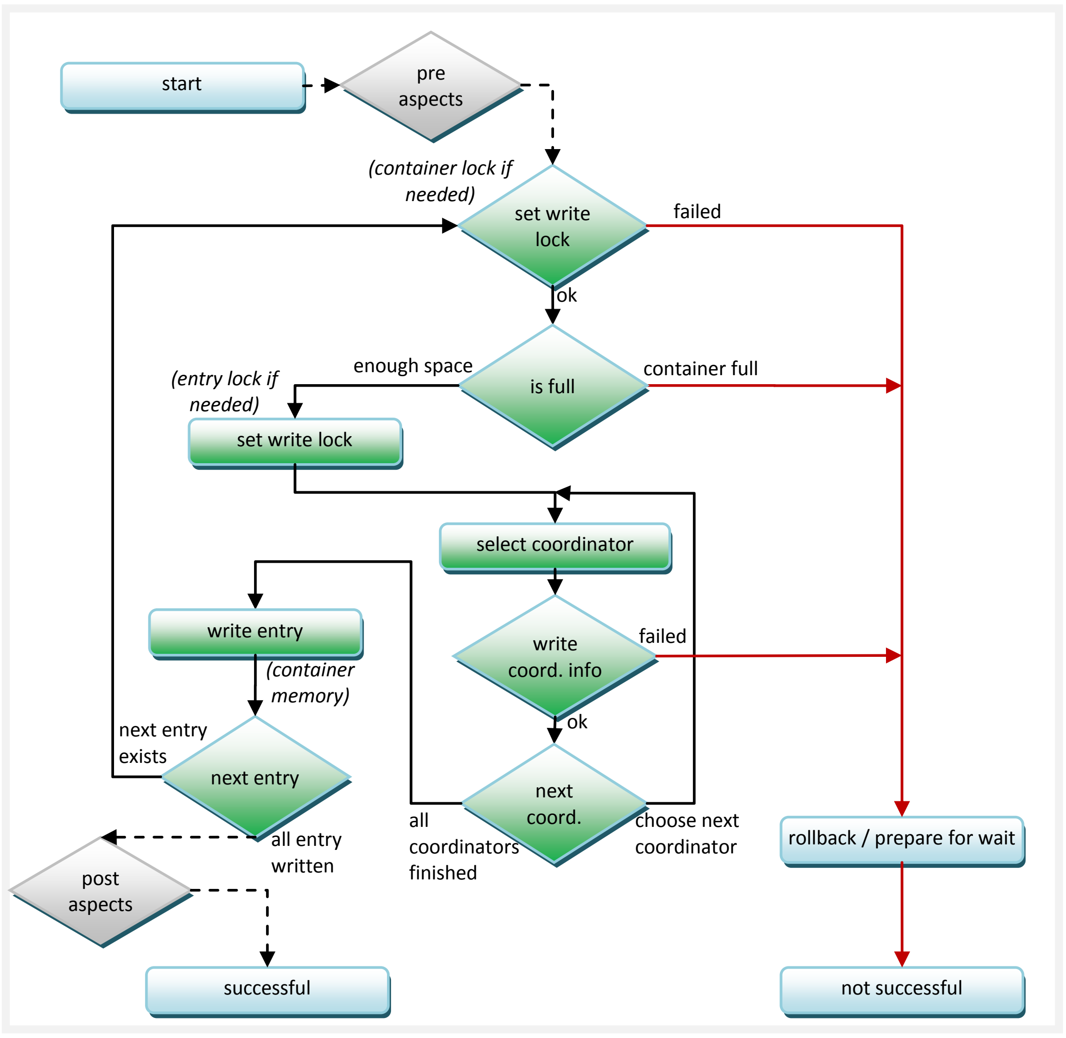


Figure 30: *Write operation on a container*

4.5.5 Processing shift

The shifting process is the most complex one of the basic operations. This is simply because the operation does not enter into waiting state when a condition is not fulfilled; for example, if the container is full or an entry with the same key is already present. To make this easier to understand, the shift process (see Figure 31) is split into three actions. This starts with the coordinator checking (shown in yellow), followed by the container is-full-prevention (in blue) and ends with the normal write process (in purple). The write process is initiated when all possible errors are resolved in the previous two steps.

```
public bool Shift(RequestWrite op, Transaction t)
```
Example 10: Signature of the shift method on the container

Directly after the successful pre-shift aspects the exclusive access starts. As in all other basic operations, the container locking will be assigned only when needed by the coordinators. Very similar to the *write* operation, every single entry will run through the whole process in the order in which they were added by the user.

The first part of the *shift* operation is the coordinator checking which is slightly different to the write operation. Here every coordinator must decide if the new entry might cause any problems and if other entries must be removed from the container. If any entries have been selected by the coordinator(s) a write lock will be requested for these entries, if not the whole container is locked.

After a successful write lock the affected entries are removed from the container. This procedure is continued for each coordinator. After this part it is assured that no conflicts with a coordinator constraint can occur.

The second part is the container-is-full prevention. The normal basic *write* operation will go to the reschedule mode when the container is full. The first check in this part is the entry boundary check on the container. If the container has enough space for the new entry the third part will be processed next, but if the container is full, at least one entry must be removed from the container. For this selection the first coordinator will be selected from the entry that should be written. If no selector is defined for the entry, the first coordinator on the container will be used. The given coordinator must decide which entry should be removed. Next, the write entry lock will be requested for the chosen entry, or if locking level is container locking, the whole container will be locked. The selected entry will then be marked for deletion from the container.

The third and last step is the *write* operation. If the two preparation steps are finished, there can be no further problem except of an unsuccessful write lock. The *write* operation itself is the same as a normal *write*. Every coordinator on the container can update its coordination information and the entry will be stored in the container.

After successful processing of all single given entries through this three step process, the exclusive access will be terminated, and the post shift aspects will be started.

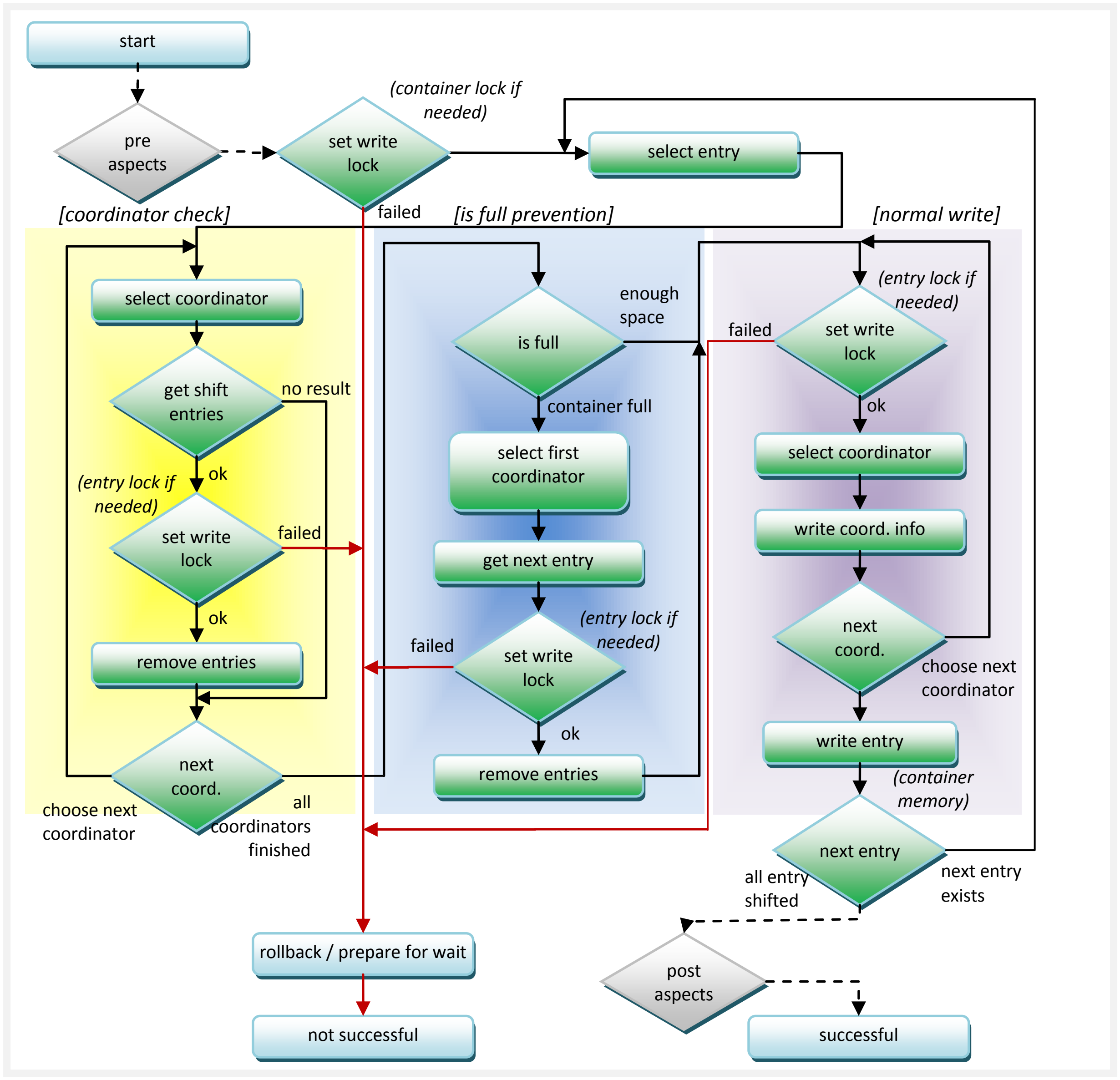


Figure 31: *Shift operation on a container*

4.6 Combination of Coordination Types

The space provides the possibilities to combine coordination types on one single container. In this discipline the space can show it real strengths in processing coordination information. For this functionality the user must specify all needed coordination types during container creation. This solution is not as flexible as dynamically adding coordination types during container usage, but the behavior of the whole container and the different coordination types is easier to understand. When a new coordinator is needed on an existing container, create a new container with all coordinators and copy all entries to the new container. The old container than can be removed.

In real life, complex problems can be solved through smart combinations of different coordinators on a single container. The possible combinations are nearly limitless. With this easy mechanism, better solutions can be created. An example is shown in the following scenario.

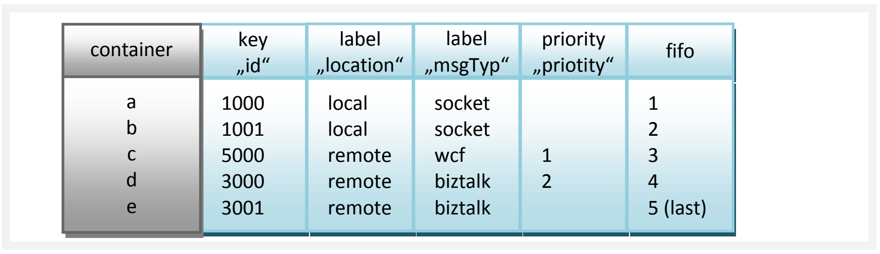
In the scenario entries in the container shall be coordinated with the following requirements:

- every message has a unique identifier so it is important to be able to access a single message very quickly
- the messages shall be distinguished into local and remote messages,
- different remote messages shall be separated for different source types
- a list shall be provided for high priority messages
- all messages shall be accessible in the order that the messages enter the container

The solution (see Figure 32) uses an unbounded container with five different coordinators. A *key* coordinator for the message identity (named “id”), a *label* coordinator for the local/remote split (named “location”), a second *label* coordinator for the source types of remote messages (named “msgType”), a *custom* coordinator (for more details see 4.7.1) for the preferred messages (named “priority”) and a *fifo* coordinator for the incoming order.

Assume that the following entries are added to the container:

- [“a”, id=1000, location=”local”, msgType=”socket”]
- [“b”, id=1001, location=”local”, msgType=”socket”]
- [“c”, id=5000, location=”remote”, msgType=”wcf”, priority=1]
- [“d”, id=3000, location=”remote”, msgType=”biztalk”, priority=2]
- [“e”, id=3001, location=”remote”, msgType=”biztalk”]



container	key „id“	label „location“	label „msgTyp“	priority „priotity“	fifo
a	1000	local	socket		1
b	1001	local	socket		2
c	5000	remote	wcf	1	3
d	3000	remote	biztalk	2	4
e	3001	remote	biztalk		5 (last)

Figure 32: Combination of coordinators on a container

4.6.1 Reading, taking and destroying entries

During the entry search – in the order that the query was built – the coordinators on the container decide which entries will be read from the container. What is important here is that every sub result is given to the next coordinator and the coordinator only reduces the entries from this subset. If the first selection returns two entries, the second one will only look at these two.

Some query examples:

- Select all entries with (location=”remote”) and (msgType=”biztalk” or “wcf”) → The result will be [[“d”], [“e”], [“c”]], because after the first selection of “remote” [[“c”], [“d”], [“e”]], the second selection uses two subselects, [[“d”], [“e”]] for “biztalk” and [“e”] for “wcf”.
- Select all entries with (id=1000) and (location=”remote”) → no result can be found
- Get all “biztalk” or “wcf” messages with priority, ordered by incoming (e.g. *fifo*) → <[“c”], [“d”]>

4.6.2 Writing entries

The *write* operation gives the corresponding selector information to the associated coordinators on the container. If one coordinator cancels the write operations because a coordinator constraint is violated, the whole operation will be rolled back and prepared for reschedule.

Some write examples:

- Write [“f”, id=1000, location=”local”, msgType=”socket”], the *write* operation will wait until the entry [“a”] is removed.
- Write [“g”, id=5005, location=”remote”, msgType=”wcf”], will be added without delay.

4.6.3 Shifting entries

As explained in Chapter 4.5.5 the difference to the *write* operation is that *shift* removes existing entries until all constraints can be fulfilled. For this elimination process the given selector order in the *shift* command is important.

Some *shift* examples:

- Shift [“h”, id=1001, location=”local”, msgType=”socket”]; this will remove entry [“b”] because both have the same key id= 1001
- If the container shown Figure 32 is bounded to five entries, then:
    - Shifting a new entry [“i”, with id=3001, location=”local”, msgType=”socket”], will remove the entry [“e”] during the coordinator check, and then [“i”] will be written to the container
    - Shifting a new entry [“j”, with id=9000, location=”local”, msgType=”socket”], if the coordinator check has no problems with the new entry, the full-prevention validation will remove one entry over the first coordinator. In this example the first coordinator is *key* and removes an entry with random selection. For example, entry [“a”] is removed.

4.7 Custom Coordinators

If the standard coordinators are not sufficient, it is possible to extend the space functionality. For this, two tasks need to be fulfilled as mentioned earlier. First the *Selector* class (namespace *XcoSpaces.Kernel.Selectors*) must be overloaded with the new functionality, and a compatible coordinator must be implemented using the *ICoordinator* interface (namespace *XcoSpaces.Kernel.Container*).

4.7.1 Example Priority Selector / Coordinator pair

In this chapter a concrete implementation of a custom selector and coordinator pair will be given. The new coordinator selects entries depending on their priority classification in the container. For this assignment a new selector must be written, called *PrioritySelector*. The coordinator shall use an explicit order and only select those entries that are tagged with *PrioritySelector* information.

4.7.1.1 Custom Selector: PrioritySelector

The main task of a selector is providing specialized information for its associated coordinator. For convenience, the *PrioritySelector* (shown in Example 11, Figure 32) provides four different constructors. The empty constructor is needed for the *microkernel* [21; 34], the dynamic loading system of the space. The second constructor with the single name parameter is for the container create phase, so many different instances of the *priority* coordinator can be active on one container. Over the name a single instance can be identified. With the third constructor a single entry can be tagged and the last constructor with the range information is for all read operations. For a shorter representation of this sample all serialization details are removed. These are needed when a serializer (e.g. binary or xml) converts the object to a serialized state in preparation for the transfer to another space instance that is not running in the same application.

```
public class PrioritySelector : Selector
{
    public String Name { get; private set; }
    public int RangeMax { get; private set; }
    public int RangeMin { get; private set; }
    public int Value { get; private set; }

    public PrioritySelector(){}
    public PrioritySelector(String name) { Name = name; }

    public PrioritySelector(String name, int value)
    {
        Name = name;
        Value = value;
    }

    public PrioritySelector(int count, String name, int min, int max) : base(count)
    {
        Name = name;
        RangeMin = min;
        RangeMax = max;
    }

    public override ICoordinator CreateCoordinator()
    {
        return new PriorityCoordinator(Name);
    }
}
```

Example 11: Implementation of the PrioritySelector

4.7.1.2 Custom Coordinator: PriorityCoordinator

The coordinator instance is a little more complex than the selector. Here the needed coordination functionality must be implemented. Example 12 shows the relevant parts of the *PriorityCoordinator* implementation that are needed for a custom coordinator.

The write operation first checks if the entry contains a *PrioritySelector*. If not, the entry will not be managed by this coordinator; otherwise the entry will be added with the helper method *AddEntrySorted*. This method inserts the new entry on the right position for fast reading access.

At the beginning of the *read* operation, the coordinator's *SelectorFits* method will be used and must approve a fitting selector (in this case the *PrioritySelector*) found in the entry selectors list. When the *read* operation starts, the difference is that the read process relies on the selected coordinator sequence. If the *PriorityCoordinator* is the first coordinator in the read process on the container, there is no list of preselected entries available, so the read process searches through the local coordination information. If preselected entries exist from a previous coordinator, only the preselected entries will be processed. The result is then a sub-select of the already given entries. For this sample implementation the read process must only select the entries when their priority level is in the range specified within the *PrioritySelector*.

The remove method searches the corresponding coordination information and removes it from the coordinator. Over the helper method *Rollback* the coordinator can inverse the deleting or adding process. This implementation always selects the first entry over the *GetNext* method when needed to select an entry to remove when the container is full. For this coordinator there are no conflicts during *shift* to be expected, so the *GetShifted* method always returns null.

```
public class PriorityCoordinator : ICoordinator
{
    …
    public bool Write(IEntry entry, ITransaction t)
    {
      PrioritySelector ls = GetSelector(entry);
      if (ls != null)
      {//check if IEntry has the correct selector
         AddEntrySorted(ls.Value, entry);
         t.AddLog(new TransactionLog(TransactionLogType.CoordinatorAdd, null,
                  Rollback, entry));
      }
      return true;
    }

    public bool SelectorFits(Selector selector)
    {
       return (selector is PrioritySelector &&
              ((PrioritySelector)selector).Name == _name);
    }

    public List<IEntry> Read(Selector selector, ITransaction t, List<IEntry>
    preSelectedEntries)
    {
      PrioritySelector ps = (PrioritySelector)selector;
      if (preSelectedEntries == null) //no preselection

           return GetElementsFromLocal(ps.RangeMin, ps.RangeMax,
                   vs.HasCountAll ? -1 : vs.Count);
      else
           return GetElementsFromPreSelected(preSelectedEntries, ps.RangeMin,
                  ps.RangeMax, vs.HasCountAll ? -1: vs.Count);
    }


    public int Remove(List<IEntry> entries, ITransaction t)
    {
       int count = 0;
       foreach (IEntry e in entries)
       {
         PriorityEntry tmp = GetEntry(e);
         if (tmp != null)
            if (_entries.Remove(tmp))
            {
              count++;
              t.AddLog(new TransactionLog(TransactionLogType.CoordinatorRemove,
                       null, Rollback, e, 0, tmp.Value));
            }
       }
      return count;
    }

    public IEntry GetNext(ITransaction t)
    {
       if (_entries.Count == 0)
           return null;
       return _entries[0].Ientry;
    }

    public List<IEntry> GetShifted(IEntry entry, ITransaction t) { return null; }

    …
}
```
Example 12: Implementation of the PriorityCoordinator

5 Transactions in the *.NET* Kernel

5.1 Introduction

For a space implementation with concurrent data access the support of transactions is very important. This means that different users should be able to access the same data concurrently. Based on the highest possible level of performance the consistency of the data must be guaranteed. This is done by the transaction concept. During a transaction a flow of operations must either be performed successfully on the space or have no effects at all.

5.2 Transactions – Theoretical

The behavior for transactions relies typically on the ACID [35; 36] properties. These properties come from database systems and stand for *Atomicity, Consistency, Isolation* and *Durability*. In the space these four are represented as follows: *Atomicity* warrants that changes will only be visible if the transaction will be committed successfully, otherwise no changes will be made. *Consistency* means that integrity constraints mustn't be violated after a transaction is finished. It is important that different transactions aren't able to interfere with each other, this feature is called *Isolation*. The last rule for transactions is called *Durability* which means that all changes must be persistent after the transaction is committed.

For the XVSM space the transaction implementation ensures that different operations on different containers can be executed as a single atomic operation. Over the transaction concept and its locking mechanism the ACID rules can be fulfilled.

The main operations on transactions are *create*, *commit* and *rollback*. Commit will be used for applying all changes permanently to the space. In contrast, the *rollback* command will revert all changes to the original state. For example, this will be needed when an error occurs or the transaction expires. After a *rollback* no changes will be made in the space; all states will be the same as before the transaction began.

The space also supports the *prepare* state for transactions. This is needed, for example, when transactions that are distributed over different cores are used. After finishing of all the tasks the transaction on the different core instances should be committed together. Before the *commit* command is sent, all transactions are prepared. This behavior is also known as *two phase commit* [37]. A transaction can be switched to the *prepare* state when all operations on the transaction are finished. When the transaction is in the *prepared* state, the space ensures that the transaction can be committed successfully. For distributed commits this is very important, so it can be ensured that every single transaction is alive and can now be committed and no other instance can make changes after this point. The space accepts only *commit* and *rollback* commands for *prepared* transactions.

The XVSM space supports *pessimistic* transactions. This means that a single transaction collects locks for every resource which is to be accessed. No other transaction can access these locked resources until the active transaction is finished through *commit* or *rollback,* except that multiple reads on the same resource are allowed. This behavior guarantees that an active transaction always is able to commit successfully, but restricts the access for other transactions. The opposite transaction handling to *pessimistic* behavior would be *optimistic* so as e.g. used in Corso [18]. Here no locks would be used during the operations performed on the transaction. At the end the *commit* procedure will check all entries modified from another transaction during the transaction life time. If a conflict is detected during this check the transaction will roll back. This would occur, for example, if transaction *A* reads an entry [“x”] and another transaction *B* removes the same entry [“x”] and *B* commits [“x”] before *A*. Then transaction *A* cannot be committed successfully because the entry [“x”] does not exist anymore and would so be rolled back.

When an operation on a container is performed a transaction is always needed. If an operation is started from the user without a transaction definition, the space will create a new temporary transaction for this operation. This automatic transaction behavior is named *implicit* transaction. These transactions will be committed directly after an operation is finished successfully, or will be rolled back in case of an error. On the other side with *explicit* transactions the user is able to define manually when *prepare*, *commit* and *rollback* should be done. *Rollback* will only start automatically when the transaction runs into a timeout. The timeout value can be defined for every single operation and additionally a timeout can be defined for the whole transaction. It defines the longest accepted period for an operation starting with the processing in the core of the space (processing start in the *CoreProcessor*, more information follows in the next chapter).

5.3 Transactions – Implementation

The transaction in the XVSM space is structured into different parts: The locking and logging part. With the locking information, the different locking mechanisms can ensure the correct access to all data. This will be shown in detail in Chapter 5. Every single instance which changes data on the container is responsible for a correct *commit* and *rollback* execution, and the information needed for this purpose can be stored in the transaction log. During *commit* and *rollback* every involved instance can access its own information for a smoothly flow. For more internal clarity a single transaction is structured into hierarchical parts as it is shown in Figure 33. Every single operation on an explicit transaction is added as sub transaction. When a new operation on a long running transaction cannot be fulfilled, the whole transaction does not need to be rolled back; only the active sub transaction must be rolled back. The rest of the transaction remains unchanged and only the single operation will be rescheduled. The hierarchical structure is only for internal use, outside of the space no nested transactions are supported.

A single log provides the answers to the three main questions: What should be changed and where? Which resources are locked? What is the root transaction? Here the information to every changed entry from a single operation can be found with the appropriate action. Additionally all locks can be found which are needed for consistent access. During the locking procedures, a transaction needs the information about its root transaction so that it is able to access all resources that have been locked by the root transaction or any other child transaction of it.

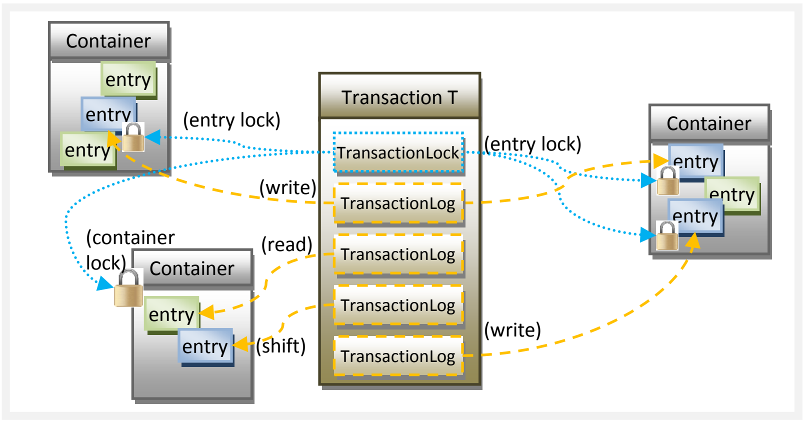


Figure 33: Example for a transaction instance with locks and logs

5.3.1 Transaction structure

Every transaction in the XVSM space is managed trough the *TransactionManager*. The class provides the main methods for *creating, preparing, committing* and *rolling back* a transaction. Additionally the transaction timeout handling can be found in this class. Every transaction in the space can be discovered over its transaction reference. A transaction reference is automatically generated in the create method during transaction creation. An additional method delivers the active transaction instance to a given transaction reference.

The *Transaction* class provides the main management functionality and access for transactions and implements the three interfaces *ITransactionLog*, *ITransaction*, *ITransactionInfo*. *ITransactionLog* provides a simple *commit* and *rollback* method. This is implemented in the *TransactionLog* class which makes delegates (function pointers) available for the two actions. Over the delegates a very flexible solution is available so every involved instance can do individual actions during *commit* and *rollback*. The interface *ITransaction* defines two maintenance methods for transactions. Over the first method *AddLog* new *ITransactionsLogs* instances can be added. This interface is used from the coordinator implementations to add information about every change in the container. The second method enables access to the root transaction for checks in the locking hierarchy. The last interface *ITransactionInfo* provides the function to receive information about all operations in the transaction, for example all added or removed entries from an adequate container. The interface is used as parameter for every *commit* aspect event. In the pre and post transaction *commit* event the user can work with the list of changes from the transaction. Beside the three interfaces, the transaction class manages the following information: the transaction timeout, a reference to the root transaction if it does exist, and the count of currently active operations that are using this transaction. The *OperationCounter* is a helper class that implements the current count of operations on a root transaction. For every single operation that is started on the transaction the counter will be incremented as soon as the operation is finished the counter will be decremented. This information is used for preparing and *commit* of a transaction. When a transaction is being prepared or committed, no operation is allowed to be active. Over the counter this information can easily be verified. For additional information about the transaction interfaces and classes a class diagram can be found in [21].

The transaction management in the XVSM space starts in the *CoreProcessor*. The *CoreProcessor* is the control center of the space to execute all different commands with the highest throughput that is possible. Here it is important that no inconsistency will be generated during parallel processing of the different tasks. For every single command the *CoreProcessor* must decide which way transactions should be handled. For more details to the architecture and the *CoreProcessor* see [21].

For the commands *read, write, container create* and *container destroy,* a helper class named *TransactionalContext* is created. In this class the transaction handling defines which type of transaction (implicit or explicit) is used. If no explicit transaction is defined, a new transaction is created otherwise a new sub-transaction is added to the existing transaction. After the initialization phase a transaction instance will exist for the next steps in the operation. The *TransactionalContext* will not be used for commands that directly manage transactions like *create*, *prepare*, *commit* and *rollback*. This behavior is also valid for aspect commands, because these commands do not work directly on containers so they do not use transactions for add and remove. The active aspect itself gets the transaction information for the current operation which the aspect is registered for.

TransactionalContext: IDisposable	
Tx	The transaction that should be used for the operation
OperationSuccess	Will be called when the operation is successful. Implicit transactions will be committed.
OperationWaitOrReschedule	Will be called when the operation goes into waiting state or is rescheduled. Implicit transactions will be rolled back.
Dispose	If the transactional context is still active (meaning the *OperationSuccess* and *OperationWaitOrReschedule* methods have not been called), the implicit transaction will be rolled back.

5.3.2 Transactions in use

In the next illustration (Figure 34) the processing of a basic operation with an explicit transaction is shown. For this example all conditions are assumed to be fulfilled so no rollback will be needed (e.g. pre aspect will be ok). After starting the processing (1) in the *CoreProcessor* of a basic operation the *TransactionalContext* (2) is created as mentioned above. In this case the transaction reference is defined and so the active transaction is selected from the *TransactionManager* (3). The *TransactionalContext* is created and a new sub-transaction (4) it is added to the existing transaction (If no transaction reference is provided in the operation information an implicit transaction will be created here). Furthermore, the operation counter in the root transaction is increased because a new operation starts. After preparing a transaction for the basic operation the requested container is selected from the *ContainerManager* (5). At this point all information is prepared for the main basic operation processing on the container. Some examples of the basic operation are shown in Figure 28 for *read* and in Figure 30 for *write*.

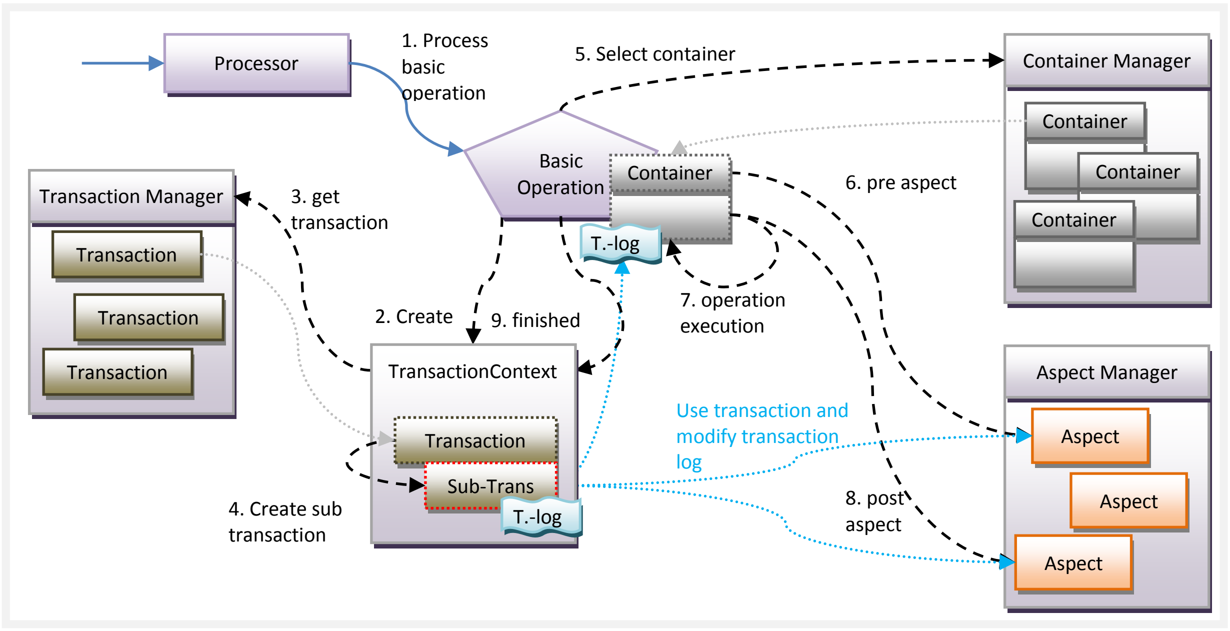


Figure 34: Successful basic operation processing with explicit transaction handling

First the pre aspect is started (6). Here only the transaction reference is provided, because the aspect itself should not get access to the active transaction itself. Here the aspect only should get the ability to create its own commands in the space with the same transaction, but should not be able to change settings directly in the transaction. This is for protection of the transaction against undesirable behavior through unexpected actions on the transaction object. After processing all pre aspects the locking procedure is started.

If container locking should be used the locking mechanism will try to lock the whole container. More details to container locking can be found in Chapter 5.4.2. Next the basic operation is started on the container (7). All entries that should be changed on the container are given to the existing coordinators. Every coordinator must ensure that all changes can be committed and rolled back correctly without any side effects. For this purpose every coordinator can write into the active transaction log and its changes are added to a protocol. Here the entry, coordinator information, the action itself (e.g. add, remove), and the *rollback* delegate can be specified. When a *rollback* is needed every single change will be reverted through the coordinator that caused it. The transaction itself has no knowledge about *commit* and *rollback* actions for every single change. It only provides the ability for processing a correct *commit* and *rollback* via delegates. If container locking is not active, the entry locking will lock every single entry. In this case the locking information of the transaction is used to gain a lock to the entry. More details about entry locking can be found in Chapter 5.4.3.

When all entries can be processed and locked successfully, the post aspect starts (8). Because the post aspect is also not allowed to change anything directly on the transaction, only the reference is forwarded. If all post aspects can be fulfilled the last action can be started in the basic operation of the container. This is a wakeup call for all operations waiting in the current transaction. Over this all waiting operations can be reactivated if they are waiting on an event like *entryAdd* like in this example. More details about the wait handling of operations can be found in [21].

During the basic operation on the container different problems can occur. Some examples are conditions for pre or post aspects, no locking is possible because the requested resources are already locked by another transaction or the coordinator task cannot be fulfilled. In all these cases the currently running transaction is rolled back automatically, the operation on the container does not return successfully and the basic operation is rescheduled.

The processing of the basic operation on the container with a transaction gives back the information if it was successful or not. If it was successful the operation count is decreased (actual action is now finished, prepare is now possible if no other operations are active) and if an implicit transaction was used it is automatically committed and the result information is returned to the user. When the basic operation result is not successful the operation is rescheduled. In this case an implicit sub transaction is automatically rolled back.

5.4 Locking

For good concurrency an efficient locking mechanism is important. Therefore the access to all resources is monitored by the locking system. If a resource should be accessed which is already locked through another transaction, the current action must wait until the lock on the resource is released. This is possible through a *commit* or *rollback* from the other transaction. The operation stays in the waiting state as long as it has not reached its operational or transaction timeout.

5.4.1 Locking interfaces and helper

The whole locking handling is managed over the interface *ILockManager* that is used in the *Container* class. In the constructor block of the *Container* class the different used coordinators decide which locking granularity will be used during the container's lifetime. If a coordinator needs container locking, it will be used, otherwise entry locking will be initialized. With the interface *ILockManager* the container only needs to specify which data should be locked (read or write) in the current basic operation. The management is implemented in the selected locking mechanism. Depending on its type the one or the other method (read and write are overloaded) will do nothing and the other will implement the locking functionality.

ILockManager	
SetReadLock	Sets a read lock for the complete container. (only for container level locking)
SetReadLock	Sets a read lock for a single entry. (only for entry level locking)
SetWriteLock	Sets a write lock for a single entry. (only for entry level locking)
SetWriteLock	Sets a read lock for the complete container. (only for container level locking)
SetCompleteReadLock	Sets a read lock for the complete container, for special situations where locking the complete container is even needed in entry level locking mode
SetCompleteWriteLock	Sets a write lock for the complete container, for special situations where locking the complete container is even needed in entry level locking mode

The interface *ILockWaitInfo* gives a clear definition why a lock must wait. This is used as decision base for read and write locks.

ILockWaitInfo	
Transaction	The transaction instance of the waiting information
Op	The operation that needs the lock
OtherContainerReadLockSet	Indicates true if another read lock than the own is set for the container

OtherContainerWriteLockSet	Indicates true if another write lock than the own is set for the container
OtherLocksSet	Uses the information in this wait object to check if there are (still) other locks set, because of that the lock for this operation cannot be set.

In the class *LockData* all locks are held from one locking implementation type.

LockData: *IDisposable*	
WriteLock	The Transaction that currently holds a write lock on the container, or null, if the container is currently not write locked
ReadLocks	The list of transactions that currently hold a read lock on the container. (The list is empty when the container is currently not read locked.)

5.4.2 Container locking

The *container* locking mechanism is a very simple opportunity to implement locking. Here the access granularity is very low, after the model complete-or-no access. Although only a single entry should be written into a container, all other transactions must wait for the release of the whole container before further action can be taken. The implementation is divided in two classes. The helper class *ContainerLockWaitInfo* implements the interface *ILockWaitInfo*. Here the important method *OtherLocksSet* defines that for a read lock no other write lock is allowed, and for a write lock neither a read nor write lock is allowed from other transactions. The second class is the *ContainerLevelLockManager* which implements the interface *ILockManager*. When all coordinators come to the decision that *container* locking should be used, this class is loaded for the locking mechanism on the container. In this class the main read and write locking methods only work on the container, the overloaded methods with *entry* locking possibilities are with no functionality.

5.4.2.1 Setting a read lock

The first action in the read lock mechanism (see Figure 35) is gathering information on the actual locking state of the container. This task searches for currently active read and write locks from other transactions and stores the results in a helper instance named *ContainerLockWaitInfo* (implements *ILockWaitInfo*). Read locks from others are seen as active when there is more than one existing read lock or the existing read lock is not from the current transaction. The write lock check is simpler because only one write lock can exist. If a write lock exists the check verifies if the write lock is from the current transaction or not. With this collected information the read lock can be managed easily.

The first condition implies no other write lock is allowed. If another write lock is active, the process will be canceled and a check for the operation time out will be performed. If the timeout is not reached, the lock request will be rescheduled. In case of timeout an exception will be thrown and the transaction be rolled back.

If the write lock is not from another transaction it will check whether or not the write lock is from the current transaction. If so, the read lock is not needed because the write lock is stronger than the read lock and the check returns success. Otherwise no write lock exists and the existing read-locks will be checked. When the root transaction does not yet exist in the read lock list, a read lock entry is added with the root transaction to the list. After adding the read lock the read lock count of all waiting locks will be refreshed. If then the waiting condition becomes invalid, the lock request will be restarted.

Then the locking event will be added to the transaction log for *commit* and *rollback*. After this the read lock is successfully set.

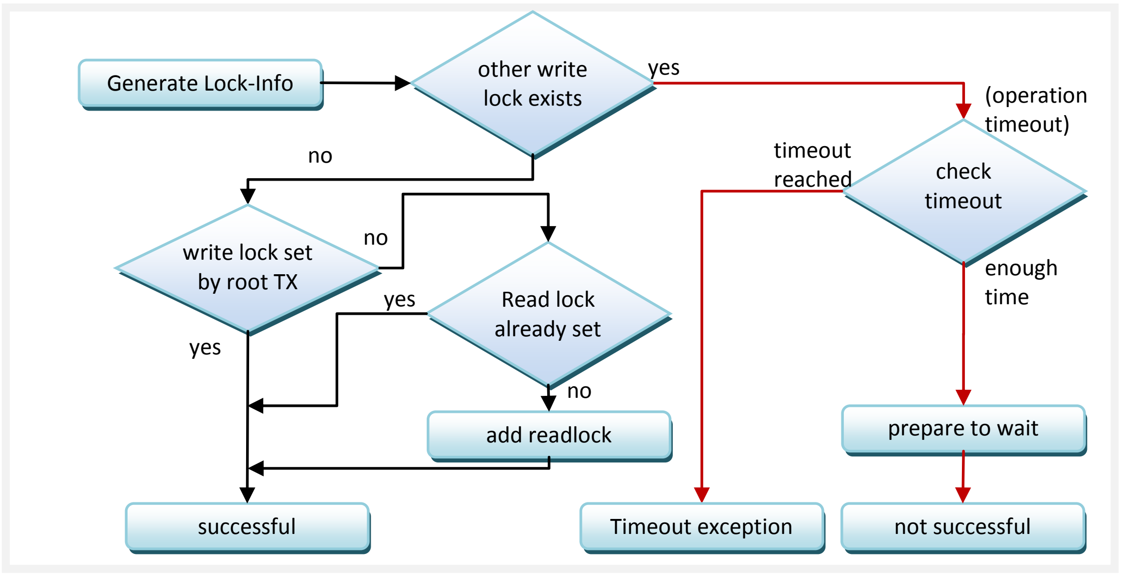


Figure 35: Setting a *read container lock*

5.4.2.2 *Setting a write lock*

The *write* lock mechanism is structured very similar as *read*. Here the same information is requested for other active locks. When other locks (write or read) are active the equal rescheduling procedure with timeout check will proceeded as with read. If no other locks are active on the container the write lock will be checked. If it is already the same as the current transaction, the task is complete, otherwise the write lock will be set to the root transaction (otherwise children of the same transaction could lock themselves). In the case where a new write lock is set, the other write lock count is recalculated for all waiting locking requests. When old waiting conditions become invalid as a result, the restart of the lock request will be initialized. As a last action the write lock event is added to the transaction log for *commit* or *rollback*.

5.4.3 Entry locking

With *entry* locking, not the entire container is locked when accessing an entry; only the requested entries are locked. This mechanism is quite more complex than *container* locking and achieves a far better performance especially when many write operations should be processed at once. Especially when the requests are spread to different entries the concurrency can be increased, and the chance that the task must wait will be reduced. In the normal case with this mechanism quite more requests can be processed concurrently as with container locking.

In Figure 36 the class diagram of the *entry* locking classes can be found. For the *entry* locking mechanism the corresponding *ILockManger* implementation is *EntryLevelLockManager*. Here are the *SetWriteLock* and *SetReadLock* methods working on single entries the ones that provide the main functionality. The overloaded write and read methods which are designed for *container* locking have no functionality. In some cases although *entry* locking is used, the whole container must be locked (e.g. for create and destroy of the container or a complete read lock for reading properties on the container, like the container's entry count). For these requirements the *ContainerLock* class uses the same locking information as for *entry* locking and works like a wrapper for all functionalities and behavior which is described in the previous Chapter (5.4.2). There is only one single but important detail, when the other locks are verified for a locking request. Here beside the whole *container* locks also the *entry* lock information must be included for verification. This means for a *read* lock that besides the condition that no other *container* write lock is allowed, also no write lock on any of the container's entries is allowed. For a write lock the expansion is similar as read, here no read or write lock is allowed, neither container nor entry lock.

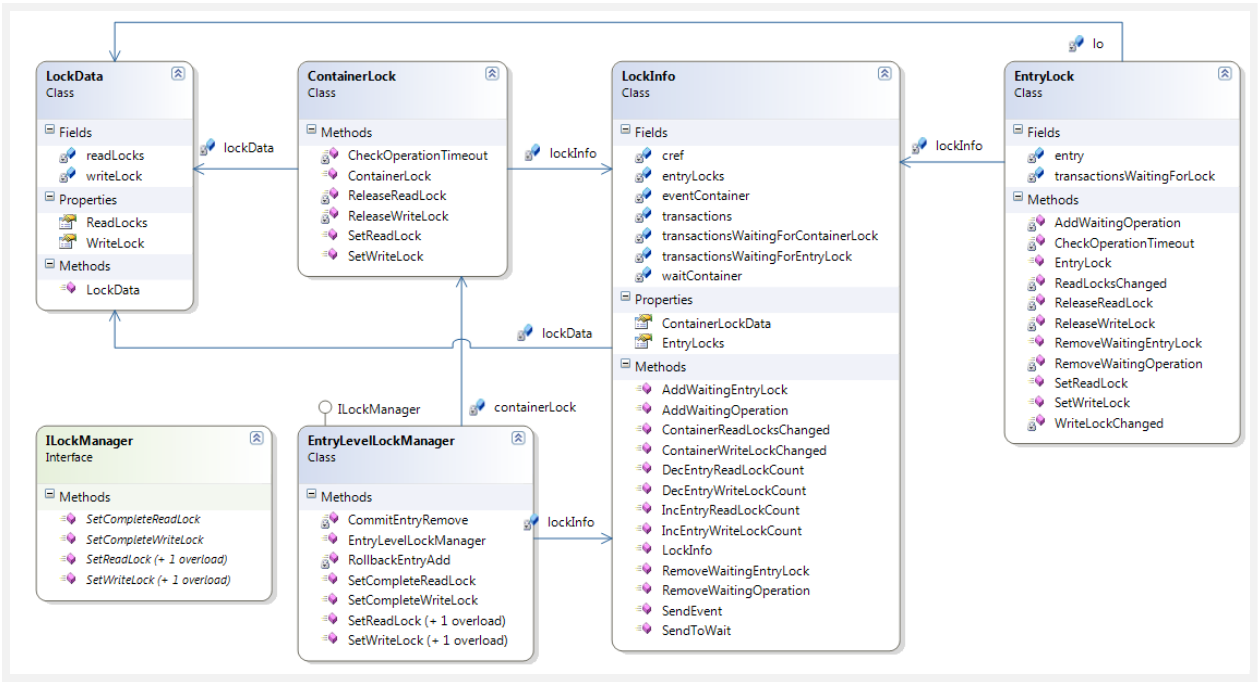


Figure 36: Class diagram of the most important entry locking classes.

The centerpiece of the *entry* locking mechanism is built by the *LockInfo* and *EntryLock* classes. In the *LockInfo* all information is managed that is needed for the lock waiting and event management at entry level locking. In the *EntryLock* class all entry locking requests are processed. The *EntryLevelLockManager* itself only distributes the different locking requests to the *EntryLock* class if the lock is requested for an entry or to the *ContainerLock* class if the whole container should be locked.

LockInfo	
ContainerLockData	The lock data for container locks
EntryLocks	The list of entry lock objects (each entry in container has an own lock object that handles the locking for this entry)
AddWaitingOperation	Creates a new wait info object(*ContainerLockWaitInfo*) for an operation waiting for a container lock and adds it to the list of waiting operations First all read and write entry locks of all other entry locks will be collected. Second all read and write locks will collected on container level
RemoveWaitingOperation	Removes an operation from the list of waiting operations for container lock
AddWaitingEntryLock	Adds a wait object to the list of operations waiting for an entry lock
RemoveWaitingEntryLock	Removes a wait object from the list of operations waiting for an entry lock
IncEntryReadLockCount	Increases the entry read lock counts of all container lock wait objects, except of the ones for the given transaction, by 1. This is

	called when an entry read lock is set
DecEntryReadLockCount	Decreases the entry read lock counts of all container lock wait objects, except of the ones for the given transaction, by 1. This is called when an entry read lock is set. Every concerned wait object is checked, and if there are no other locks set, an event is sent to wake up the waiting operation
IncEntryWriteLockCount	Increases the entry write lock counts of all container lock wait objects, except of the ones for the given transaction, by 1. This is called when an entry read lock is set.
DecEntryWriteLockCount	Decreases the entry write lock counts of all container lock wait objects, except of the ones for the given transaction, by 1. This is called when an entry read lock is set. Every concerned wait object is checked, and if there are no other locks set, an event is sent to wake up the waiting operation.
ContainerReadLocksChanged	Called by whenever the container read locks are changed. All waiting operations are checked, if they can be woken up.
DecEntryWriteLockCount	Called by whenever the container write locks are changed. All waiting operations are checked, if they can be woken up.
SendEvent	Sends an event for the operation that is waiting with this *waitInfo* object. (The event is written to the event container.)
SendToWait	Send an operation into waiting state by writing it into the wait container

5.4.3.1 *Setting a read lock*

Over the interface *ILockManager* the *EntryLevelLockManager* gets the read lock request for a single entry from the container. In the first step the lock information for the given entry is chosen. The lock information is generated when the entry is added to the container (during the first write). If no lock information of the entry is available an exception will be thrown, otherwise the normal processing will be continued. The set read lock request is forwarded to the *EntryLock* instance.

Here also the locking information gathering will start for other active locks as in the *container* locking level. The following information is collected in an *EntryLockWaitInfo* instance:

EntryLockWaitInfo: ILockWaitInfo	
Transaction	The transaction for which the lock should be set
Op	The operation that needs the lock
IEntry	The entry that should be locked
OtherEntryReadLockSet	This property will be true if more than one entry lock active or the only existing entry read lock is from another transaction
OtherEntryWriteLockSet	This property will be true if an write lock exists and the lock is from another transaction
OtherContainerReadLockSet	This property will be true if the same conditions will be fulfilled on container level (*OtherEntryReadLockSet*).
OtherContainerWriteLockSet	This property will be true if the same conditions will be fulfilled on container level (*OtherEntryWriteLockSet*).
OtherLocksSet	For write lock every other read or write lock on both locking levels (entry/container) returns true. For read locks no other write lock will be accepted (entry/container)

When all this information has been evaluated the decision process is started. If other locks are currently active the operation timeout is validated. If the operation timeout is expired an exception is thrown (*XcoOperationTimeoutException*) and the roll back sequence for the transaction is started. Otherwise the operation is scheduled. If there are no other conflicting locks, the process is continued.

If a write lock exists on the container and it is from the current root transaction, the read locking is not required anymore, because the existing write lock is stronger than the requested read lock. (If a write lock exists and it is not from the current root transaction the request will be directly scheduled). Then the search for already existing read entry lock is started. If a read lock is already active no further action is required, otherwise the root transaction is added to the read lock list. After adding a new read lock for all waiting requests the entry read lock count is updated and requests that are no longer blocked are woken from the waiting state. Additionally the entry read count on the current transaction is increased, then all locking information is refreshed and the read lock event added to the transaction log, and the read lock request is successfully finished.

5.4.3.2 Setting a write lock

The write lock also starts like the read lock with information gathering and building the *EntryLockWaitInfo* helper object. When other locks (read/write) are found that are not from the current transaction, the operational timeout check will be processed like in the read procedure. If the operational timeout is not yet reached the lock request will be rescheduled.

If no other locks are currently active, the existing write lock is validated. If the write lock is owned by the current transaction the locking process is finished. Otherwise the write lock is set to the root transaction. Then all information is refreshed as in the read lock procedure. First the entry write lock count of all waiting locks is recalculated. When then waiting conditions are released, the lock request will be restarted from the waiting state. As one of the last tasks the entry write lock count for the own transaction is incremented and the write lock count for all other waiting locks not in the current transaction are refreshed. The final task adds the write lock event into the transaction lock for *commit* and *rollback*.

5.5 Deadlock Detection

A deadlock occurs when two different concurrent transactions attempt to access the resources that are already held by the competing transaction. Both transactions will then wait for the other to release the lock from the resource.

For deadlocks four conditions are necessary to occur, *mutual exclusion, hold and wait, no preemption, circular wait*, also known as the Coffman conditions [38]. *Mutual exclusion* implies that a resource cannot be used more than once at the same time. *Hold and wait* means that a transaction which already holds resources may request new ones. *No preemption* states that an already held resource cannot be automatically removed from the transaction. The resource can only be released over an explicit action of the transaction. The last condition is *circular wait* which defines that a scenario is possible where two or more transactions build a circular chain where each process waits for the release of resources from the other next in the chain.

All these conditions can be shown in a simple example for a deadlock in the next illustration (see Figure 37). Here two different transaction “I” and “II” want to read and write to two different containers “a” and “b” using *container* locking. First transaction “I” gets the container lock for container “a”, at the same time the transaction “II” ensures the container lock for “b”. Then transaction “I” wants to access container “b” but it is already locked through transaction “II” vice versa the transaction “II” wants to access container “a” that is locked through transaction “I”. Then the deadlock is complete, both transactions cannot fulfill their operations and will wait endlessly.

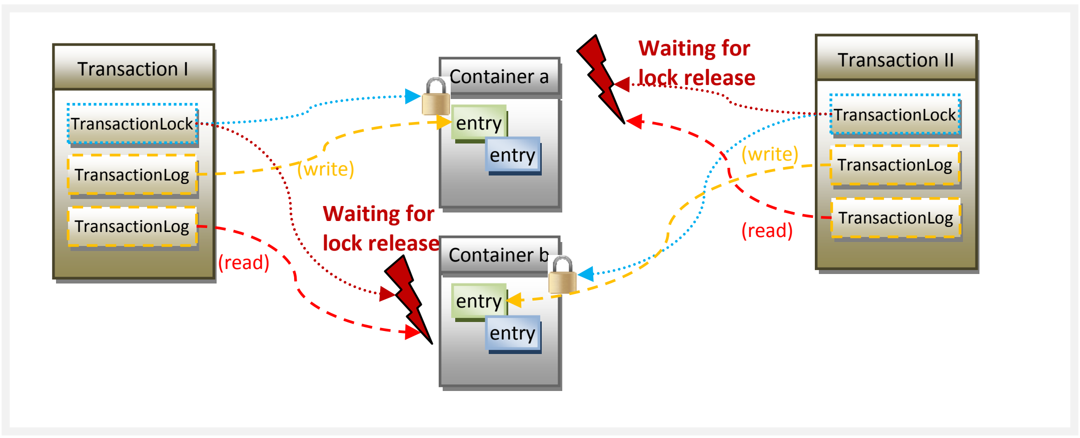


Figure 37: Example for a deadlock

5.5.1 Limitations of the Core

Deadlock prevention is a complex topic. Here a solution would be needed that one of the four conditions can not be possible so that no deadlock will occur. As long as pessimistic locking is used in the XVSM space the mutual exclusion will be needed and cannot be solved over non-blocking algorithms. To prevent the hold and wait condition every transaction would have to specify all required resources before the transaction starts. But implementing this functionality is not easy and not very flexible. Large transactions could only start when all information is available which resources are needed, and often this will not be possible because in a transaction the user will react on data which was read in the current transaction. A solution for the no preemption condition may be also very difficult when pessimist locking is used. Here an algorithm would have to decide which transactions are allowed to get the lock to the resource and which transactions must be rolled back and rescheduled. This behavior can cost a large amount of performance. To avoid circular waits a transaction is allowed to wait for resources, but ensure that the waiting cannot be circular. One solution might be to create a hierarchical structure between resources for example with precedence. A transaction can only request resources in order of the precedence. If resources are already held only resources with higher precedence can be requested.

Deadlock detection itself is may be easier to implement than prevention solutions. Here an algorithm must track all allocated resources and transaction states. When a deadlock is identified the algorithm must specify which transactions should be rolled back and rescheduled. For deadlock detection some additional problems exist. How often should this algorithm start? How many resources and how much performance does the deadlock detection itself require? Which transaction should be rolled back (what is the best decision in this case)? The next problems come up when distributed transaction are used.

5.5.1 Solution

The solution for the deadlock problem in the current implementation of the XVSM space is composed of specified timeouts for every single transaction. After the timeout is expired a transaction will be automatically rolled back. With this solution, endless waiting situations are not possible unless the user specifies an infinite timeout. So in this case it is up to the user to define meaningful timeout values for his/her transactions so that deadlocks will be prevented.

6 Remote communication

6.1 Introduction

When a XVSM core instance shall not only work in embedded mode a remote communication mechanism is needed. Instead of supporting only a single transport protocol, an open extensible solution is preferred to add the transport service the user requires. For this purpose the XVSM space has clear and straightforward communication contract by which the XVSM core is able to communicate to the outside. For more general information on contracts in the XVSM space see [21]. In the following illustration (Figure 38) classes of the *CommunicationService* contract are shown.

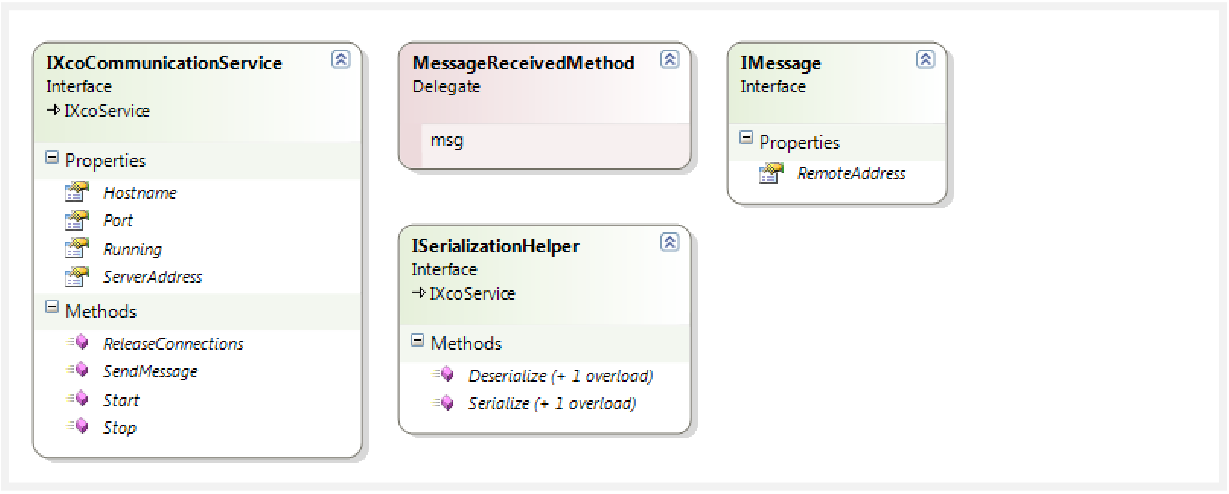


Figure 38: The important classes of the *CommunicationService* contract.

The *IXcoCommunicationService* interface allows the XVSM core to transfer messages over a remote channel without having any knowledge about implementation details of the underlying transport protocol. On the other hand, the communication service itself must only deal with the message and make sure that the messages will be delivered and received. This decoupling is important given that additional communication services should be easliy supported by the space. The public properties of the interface should give an overview of the state of the service. With the *hostname* und *port* information the local address is generated. In the running state, the space can check if the service is already active. The *ReleaseConnections* methods allow the communication service to clean up all active connections before a service is shut down and the underlying communication instances are disposed of and cleaned up.

During the start sequence two important pieces of information are handed to the communication service. First the delegate for received massages is specified; every message that is received over the communication server will be forwarded to the defined delegate. The second parameter in the start method is the *ISerializationHelper* instance. Using this inferface, the communication service can serialize and deserialize the message instances. In the current implementation of the serializer the developer can decide which way the messages should be converted from the object instance to a byte array and vica versa. Through the usage of the *ISerializationHelper* the developer does not need any knowledge of how the data should be serialized in the communication serivce, but only needs to concentrate on the technical details for the new communication service specification.

IXcoCommunicationService :	IXcoService
Hostname	The hostname on which the service is running
Port	The port on which the service is running
Running	True if the service is running
ServerAddress	The address of the server
ReleaseConnections	Releases all current active connections
SendMessage	Sends an instance of *IMessage* to the given address
Start	Starts the communication service with IP address and port. Additionally the method delegate for receiving messages (*MessageReceiveMethod*) and the serializer instance (*ISerialzationHelper*) can be defined here.
Stop	Stops the communication service

6.2 Processing a message

In the next chapter the workflow for remote requests is shown. New requests to be processed by the space can be generated over the embedded API or via network using the communication service. For more information about message handling in the core see [21].

For the next sub chapters, no exception handling will be mentioned. When something goes wrong the space will throw an exception. Here not only a standard exception will be thrown; instead a specialized XVSM exception will be generated with the most possible available details. More details about the different exception types can be found in [21]. This concept is not only valid for local exceptions; if an exception occurs during processing on a remote space the exception will be caught and sent to the defined remote address.

In the standard space communication concept the single cores do not connect to other core instances. They only send a single message over the space with no connections. The whole space should be shown as a coherent system. For the normal developer it makes no difference where the real data is hosted. The communication service automatically manages the communication in the background. For most implementations (like *TCP* communication service) a connection is needed in the background and this connection should be held for a certain amount of time. For example to open a new *TCP* connection for every single message does not make a lot of sense because the *TCP* handshake needs a considerable amount of time. This is especially important when a large amount of small packages need to be sent. Managing open connection consumes more memory, but the performance will be enhanced. When the connection is not used for a configurable time span, the connection will be closed and the cleanup begun.

6.2.1 Sending a remote request

In the next illustration (Figure 39) the processing of a request is shown. In this case a TCP (Transmission Control Protocol [39]) communication service is configured for the space. A simple example is that a request from *core A* is created and then adds a new entry into a container on *core B*. The start procedure for all requests to be processed is the same, and it makes no difference if it is to be processed on the local or on a remote space. First a new message instance is generated with a message identifier (short message id) that is unique for this core instance. Over the message id the core can identify the response after the processing. Then the operation context of the message is checked. The operational context additional information can be provided for a single message. This can be used, for example, when security tokens are needed to authenticate the current request on the other space. The space allows defining default operational context details. If a default context is set, then all messages will receive this information, otherwise a default empty object will be generated.

As next step in the sequence the remote information is updated. Here the system must decide if it needs remote communication or not. For remote communication the given remote address must be defined in the message, which must of course be different to the local communication service address. If remote sending is required the content of the message is serialized by the currently active serialization helper instance. More information about serialization can be found in Chapter 6.4. When the serialization is finished, the timeout is calculated. If a timeout for the operation is specified it will be used, otherwise a default timeout (30 seconds) will be set.

For the timeout management a waiting-object is created in which all important information, such as message id, remote address and timeout is held.

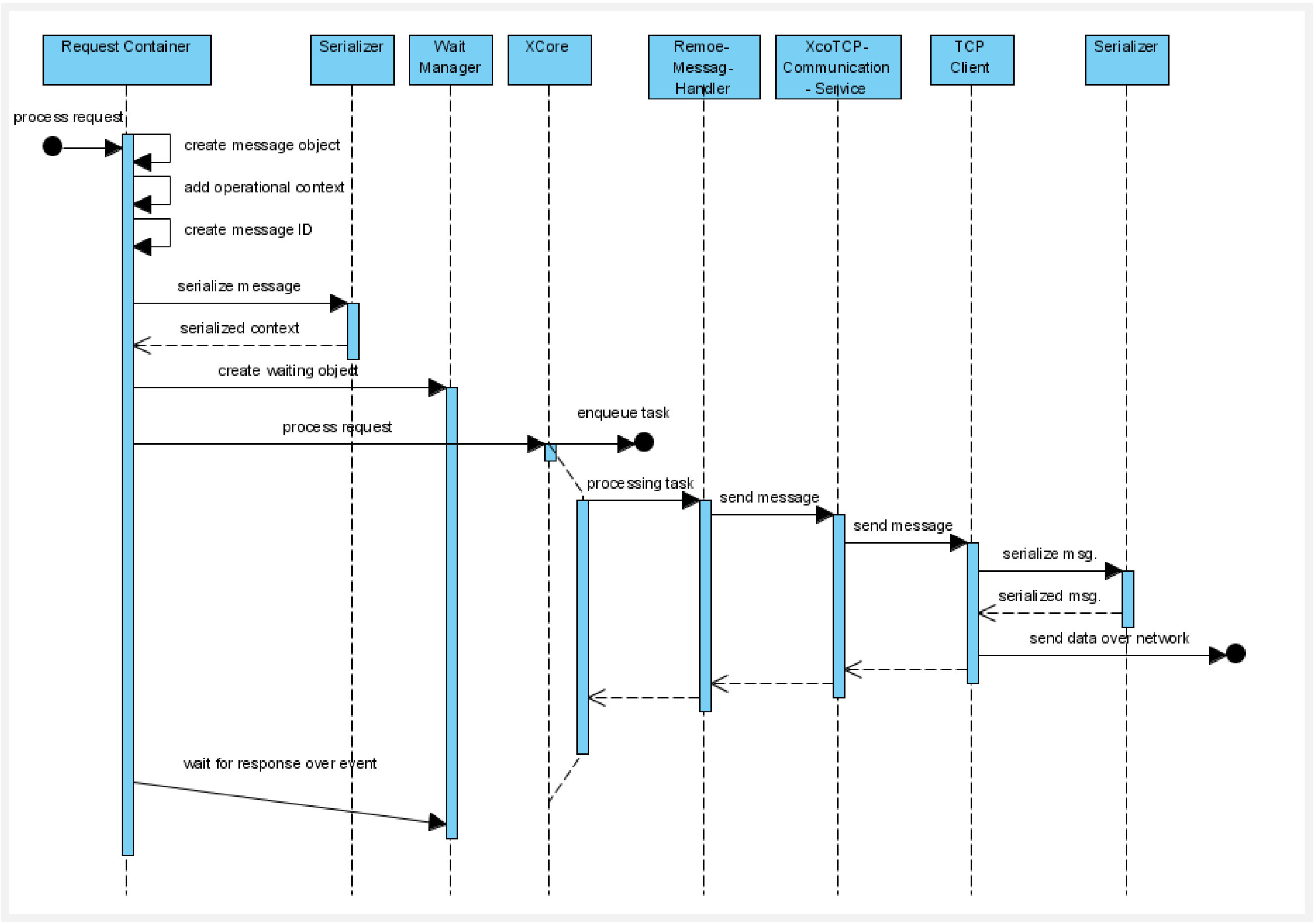


Figure 39: Processing of a request over remote communication part I

At this point, every prepared task is forwarded to the *XCore processor*. The *XCore processor* adds the task to its processing queue and will start processing the task. With this last action the start procedure is nearly finished. Immediately after forwarding the task, the start procedure falls into a waiting state. An event from the waiting management reactivates the task when the response is available or the operation timeout is reached.

The processing itself is continued over the scheduling mechanism of the *XCore processor*. As soon as there are free resources the task will be processed. If the task is to be processed locally the internal request management will start here, otherwise the message will be given to the *RemoteMessageHandler* instance. This class is the intermediary between the core classes and the currently active communication service. The *RemoteMessageHandler* not only manages the transfer of the messages in both directions, but is also responsible for the communication service start and stop sequences. In this example the *Send* method hands the message over to the active communication service.

In the current example the active communication service is the *XcoTCPCommunicationService*. This creates a new *TCPClient* class instance if no active connection to this remote address exists, otherwise the already existing one will be used. In the next step the remote address is changed to the address of the currently active communication service. With this small change the remote core knows which host the response should be transferred to, and the message is forwarded to the *TCPClient* instance. The *TCPClient* itself serializes the message before the transfer can be started. Directly after this process, the transfer over the network starts.

6.2.2 Receiving a remote response

Corresponding to the last chapter, here the flow of a response message from another core instance is shown in Figure 40. When the other core instance receives the processed request the response will be transferred. In this case the *TCPServer* is the receiving part of the *XcoTCPCommunicationService* implementation and is waiting for new connections from the outside. The accepting of new connections is decoupled from the processing and is managed over a thread pool. When a new data object can be read from an already running receiving task this object is deserialized with the active serializer instance. If the result after the deserialization can be parsed to an *IMessage* object the new instance is returned by the message received delegate. Normally here the *MessageReceived* method will be called from the *RemoteMessageHandler* instance. During the startup of the communication service, the *RemoteMessageHandler* defines the delegate for received messages. The *RemoteMessageHander* passes the response message back to the XCore *processor*. Here the corresponding waiting object is found. The response object is then added to the waiting instance and the signal is generated to release the waiting state.

The closing action is that the waiting instance is reactivated. The first action is to deserialize the content of the message. If an error was thrown on the other space instance the content of the message will contain an exception. When an exception is found in the content, the exception will be thrown, otherwise the content of the message will be returned and the processing of the message will be finished afterwards.

If no response is returned before the operation timeout is reached, the waiting operation is canceled and a corresponding timeout exception is thrown.

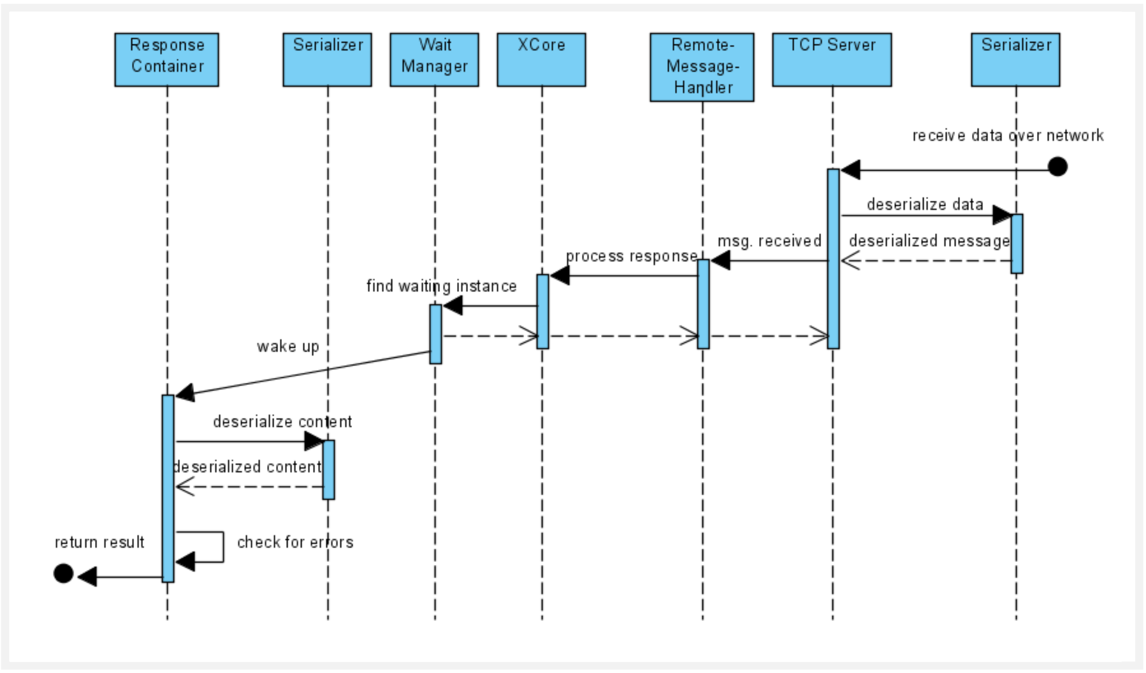


Figure 40: Processing of a request over remote communication part II

6.3 Types of communication

In the next chapters the different supported implementations are presented. Every single variant has its own advantages and disadvantages. The communication implementation can be divided into two groups. The first group works very well in local area networks and therefore the performance is very high, but if communication over firewalls is needed, every firewall must be configured manually. The *TCP* and the *WCF (Windows Communication Foundation)* implementations belong to this group. The second group on the other hand is designed to communicate without any problems across firewall borders. This is done over tunneling via servers in the internet and so the performance is not as high as with local communication, but instead the communication works over the internet. Currently the *BizTalk* services implementation is the only option; a solution using jabber will be implemented in a later version.

6.3.1 TCP

The *XcoTCPCommunicationService* allows the XVSM space to communicate over sockets. This implementation is normally used when the communication needs to be platform independent and work together with the *JAVA* implementation (*Mozartspaces* [33; 13; 40]) of the XVSM space.

For the execution the service has two main classes. The first is the *TCPServer* class which manages all incoming connections. The second class, which is the *TCPClient,* is responsible for sending the messages over the network. Besides the main functionality to send and receive messages, all open connections are managed by this service.

In the next illustration (Figure 41) the normal behavior of the *XcoTCPCommunicationService* is shown. For better performance to a single endpoint (combination of IP-address and port) only one connection at a time will be established and held open. When a new message is to be delivered to another XVSM core, a *TCPClient* instance is needed. If an active *TCPClient* does not already exist, a new client will be created. To create a connection to another XVSM core it is necessary that the firewall rules are correct and that the other space instances can be connected directly.

The *TCPClient* itself connects to the other XVSM core during the initialization phase. In the message send method the message serialization over the given active serializer is started. After the message is converted to a byte array, the data is sent over the socket. All TCP client connections that are not needed for more than a minute will be closed and cleaned up automatically.

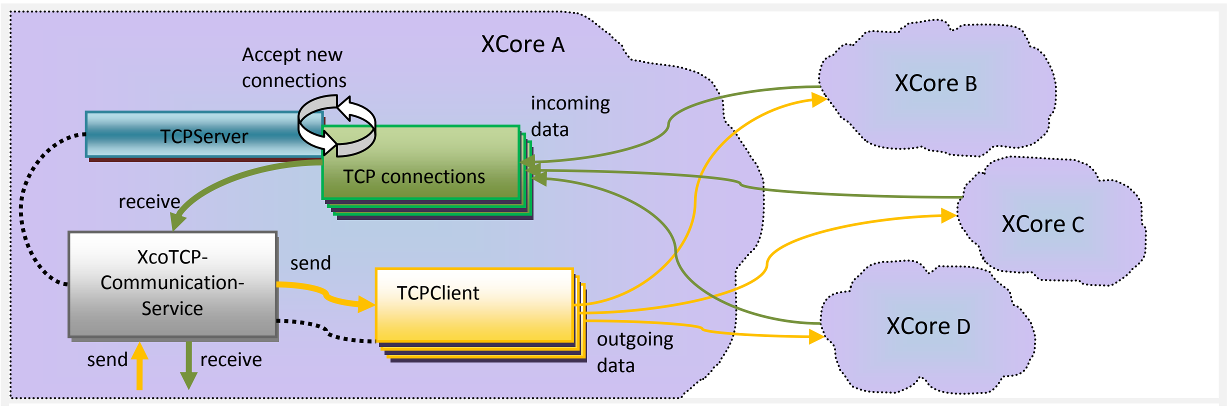


Figure 41: The Standard *XcoTCPCommunicationService*

On the other side messages are received over the *TCPServer* instance, which accepts new connections on the given port. To ensure no performance issues arise from concurrent processing of all incoming requests, all requests are managed over a thread pool. Before the processing of an incoming connection is started, the connection itself is added to the active incoming connection pool. When the service is to be stopped, all current active connections must be closed before the socket is closed.

In the processing loop the system waits for new message objects. Every incoming object is directly deserialized with the given active serializer. When the instance is an *IMessage* instance the message is forwarded over the receiving delegate to the processing mechanisms of the core.

A special scenario is shown in Figure 42, here a firewall is situated behind the master space (XCore A). The other space instances (XCore B-D) are e.g. running on mobile devices. On mobile devices the mobile core should not open a port due to security and performance issues. For this behavior the *TCP* service can be switched to the bidirectional mode in the settings. When the bidirectional mode is active on a space instance, no direct connections are established (connecting directly to a mobile device is not possible). Instead, outgoing data is managed over the same connection as incoming data. Here all mobile space instances have to connect to the master space on a well known port, which is open on the firewall. If a mobile space instance (XCore B-D) has established a connection to the master space (XCore A) the connection will be held as long as possible. This has two simple reasons. First the master space can communicate over the existing connection with the mobile space. The second reason is economically motivated. As long as no data is transferred over the active *GPRS* connection no additional costs will be incurred. In contrary the reconnect into the *GPRS* network will generate new cost for the operator of the software, therefore a connection will be held as long as possible.

The bidirectional mode requires some little changes in the *TCPClient* and *TCPServer* module. The processing of newly connected clients in the *TCPServer* requires an extension. Directly after reading the first message, the internal remote address is read, and with the remote address, the connection is added to a hash table of existing connection. This connection-collection is used when a new message is to be sent. The corresponding active connection can be found in the hash table via the destination address in the message. This connection is also used for sending of new messages to the other space instance instead of establishing a new connection. The last difference between this and the normal mode is in the timeout handling. Normally all *TCPClient* instances for sending are closed when no message has been sent within the timeout period. In the bidirectional mode only the *TCPClient* object is removed and destroyed - the underlying connection is not closed. This connection must be closed off from the space situated outside the firewall.

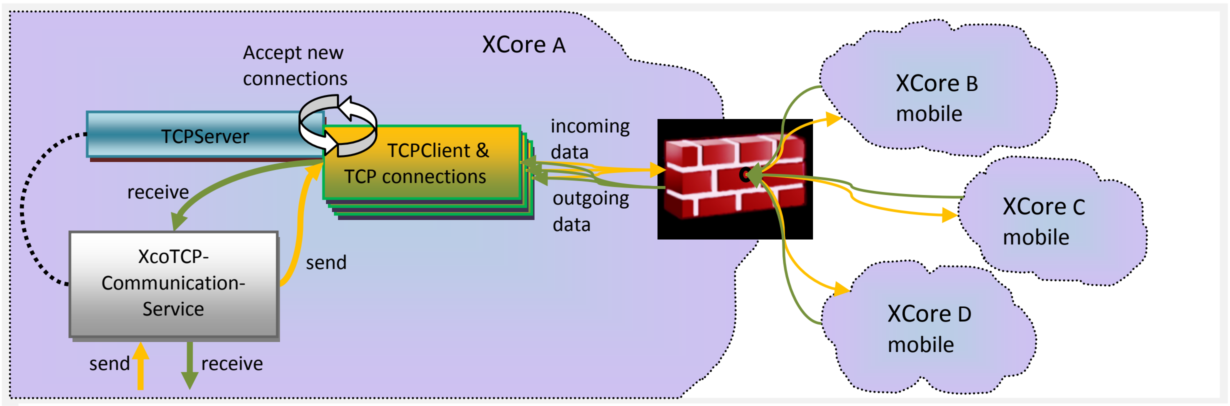


Figure 42: The *XcoTCPCommunicationService in bidirectional mode*

6.3.2 WCF

6.3.2.1 *Introduction*

The *Windows Communication Foundation* (*WCF*) [41] is one of the most important new parts of the *.NET Framework 3.0*. *WCF* unifies different communication technologies (*DCOM* [42], *Enterprise Services* [39; 43], *MSMQ* [44], *WSE* [45] *and Web-Services*) over a single and common service oriented at a programming model for communication. The main field of *WCF* is in supporting service oriented architecture principles for distributed services.

In *WCF* a service is accessed via an endpoint. The endpoint is composed of **a**ddress, **b**inding and **c**ontact, also known as *ABC* principle [46]. The address defines where the service is hosted. In the binding the supported protocol (*HTTP*, *TCP*, *UDP*,…) and all other communication details (e.g. timeouts, serialization, security definitions and so on) can be set [47]. In the contract the interfaces of the service are specified for the client.

6.3.2.2 *WCF in the XVSM space*

The *WCF* service implementation (*XcoWCFCommunicationService*) is the standard communication service in the *XVSM* space. That is because *WCF* provides communication functions on a very high level and is easy to use and implement. The main reason lies in the very powerful configuration possibilities of *WCF*. In the *XVSM* service the contract is defined and all settings for the protocol and communication details can be specified in the configuration file. For most settings, no changes in the source code are needed.

As mentioned in the previous chapter, the open connections are automatically managed and reused for more than one message. By communicating via *WCF* the communication slows down when the connection is repeatedly closed and reopened.

For the *XcoWCFCommunicationService* the *WCF* contract is shown in Example 13. The interface *IRemoteSpaceService* is quite simple – with the *connect* method a new connection is established. Via

SendMessage new data can be transferred, and when all work is done the connection can be released with *Disconnect*. The *SessionMode* is responsible for the connection handling and specifies, when the connection must be initiated and terminated in the corresponding parameters.

```
[ServiceContract(SessionMode=SessionMode.Required)]
public interface IRemoteSpaceService
{
  [OperationContract(IsInitiating = true, IsTerminating = false)]
  void Connect();

  [OperationContract(IsInitiating = false, IsTerminating = true)]
  void Disconnect();

  [OperationContract( IsInitiating = false, IsTerminating = false)]
  void SendMessage(byte[] xCoreData);
}

public interface SpaceServiceChannel : IRemoteSpaceService, IClientChannel { }
```
Example 13: Contract of the XcoWCFCommunicationService

The class *SpacePortal* implements the interface *IRemoteSpaceService*. The only method that must be overridden with new functionalities is the *SendMessage* method, the other two (connect and disconnect) can be left empty because *WCF* automatically handles the rest.

During the startup phase of the *XcoWCFCommunicationService* the setup of the *ServiceHost* is started in a background thread. The *ServiceHost* mechanism of *WCF* allows hosting a contract over different endpoints with different bindings. In the case of the communication service, the *SpacePortal* is used as contract. First a valid *WCF* endpoint is generated with the hostname of the machine. Then the configuration is checked. If an application configuration file exists the important values for the *WCF* service are searched there. When a valid configuration block is found, these settings are used, otherwise a default *TCP* binding is generated. With the endpoint address (already defined), binding (now loaded) and contract (*IRemoteSpaceService*) all preconditions (ABC principle) are fulfilled and the hosting of the *SpacePortal* can begin.

After this point new connections are handled over the *WCF* service. This means that every data object transferred over the network to this hosted service can be handled in the *SendMessage* method of the *SpacePortal*. Here the byte array of data should only be deserialized to an *IMessage* instance. The *IMessage* instance then will be forwarded over the delegate to the core processing mechanisms.

For sending new messages over *WCF* a new *RemoteSender* is generated when no currently active instance is registered for the current remote address (= destination address of the message). Similar to the *SpacePortal* initialization sequence the *RemoteSender* can also be defined via the settings file. When no configuration block is found, default settings for a *TCP* binding are added. A *WCF* channel can be established to a *SpacePortal* instance via a *ChannelFactory<SpaceServiceChannel>*.

After the *RemoteSender* instance is initialized the connect method will be called. The *WCF* abstraction on this channel only allows the three methods from the contract for the communication with the other end of the network. The connect method of the channel will be invoked by the connect method of the *RemoteSender*. To send the message, the object will be serialized and the resulting byte array will be sent over the *WCF* channel.

The *WCF* takes care of many details which a developer must otherwise consider when the communication would be managed manually. To obtain the best results, the

XcoWCFCommunicationService additionally manages some *WCF* details for the user. First the active connection may be reused and may also be cleaned up when the service is stopped. If a connection breaks or a timeout occurs the dead connection will be removed automatically and used resources will be released. Similar to the native TCP implementation in the previous chapter a connection timeout handling exists which closes *RemoteSender* channels that are not used for a defined amount of time.

The current configuration is tested for the *WCF* bindings for TCP and *Named-Pipes* [48]. The *WCF* naming is *netTcpBinding* and *netNamedPipeBinding*. Named pipes allow simple communication between a pipe server and one or more pipe clients. This can be especially useful for communication between related or unrelated processes with security checks. The communication can also run across machine borders over the network.

The configuration does not work with *Microsoft Message Queues* (MSMQ) [44] because the contract is not compatible with the *netMsmqBinding* definitions (a message queue normally sends the data one way and doesn't have any open connections). Microsoft Message Queues also allow exchanging messages when the other process is not currently running. The messaging system supports guaranteed message delivery, security and priority-based messaging. When the other partner is ready again, the queued messages can be processed.

In the next illustration, (Example 14) an example configuration for the *XcoWCFCommunicationService* with *netTcpBinding* can be found. If *netNamedPipeBinding* is used only the binding tag must be changed from *netTcpBinding* to *netNamedPipeBinding*. With this simple change the service then runs with another communication protocol.

```
<system.serviceModel>
<services><!-- For XVSM cores server -->
  <service name="XcoSpaces.Kernel.Communication.WCF.SpacePortal"
         behaviorConfiguration="XCoreBehavior">
    <endpoint name="XVSMDefault" address=""
           binding="netTcpBinding"
           bindingConfiguration="XVSMBinding"
           contract="XcoSpaces.Kernel.Communication.WCF.IRemoteSpaceService" />
  </service>
</services>
<client> <!-- For XVSM core clients -->
  <endpoint name="XCoreClientDefault" address=""
           binding="netTcpBinding" bindingConfiguration="XVSMBinding"
           contract="XcoSpaces.Kernel.Communication.WCF.IRemoteSpaceService"/>
</client>
<behaviors>
  <serviceBehaviors>
    <behavior  name="XCoreBehavior">
      <serviceThrottling maxConcurrentSessions="10000" />
    </behavior>
  </serviceBehaviors>
</behaviors>
<bindings>
  <netTcpBinding>
     <binding name="XVSMBinding"  maxReceivedMessageSize="512000"
              maxBufferSize="512000">
       <readerQuotas maxArrayLength="512000"   />
       <security mode="None" />
     </binding>
  </netTcpBinding>
</bindings>
</system.serviceModel>
```
Example 14: Configuration example for netTcpBinding

6.3.3 BizTalk Services

The *BizTalk Services* [49] were an initiative from Microsoft to build a lightweight programming model to connect applications over the internet. During our space development the *BizTalk Services* were a *Community Technology Preview* (CTP) and nobody knew precisely whether this new software module would come to a release state and in which product Microsoft would present it. (Meanwhile the BizTalk Services are included in Microsoft's new *Windows Azure* platform [50])

All applications that need to communicate with each other must do this over the new *Internet Service Bus* (ISB) (see Figure 43). This is light-weight implementation as known from bigger, already existing Enterprise Service Bus implementations specialized for internet service needs. The Internet Service Bus has automatic identity, authentication and authorization mechanisms for a secure communication between partners. The ISB also supports syndication, callbacks, notifications and a lot more. All these functionalities are integrated to a well known Microsoft *WCF* binding [48] mechanism.

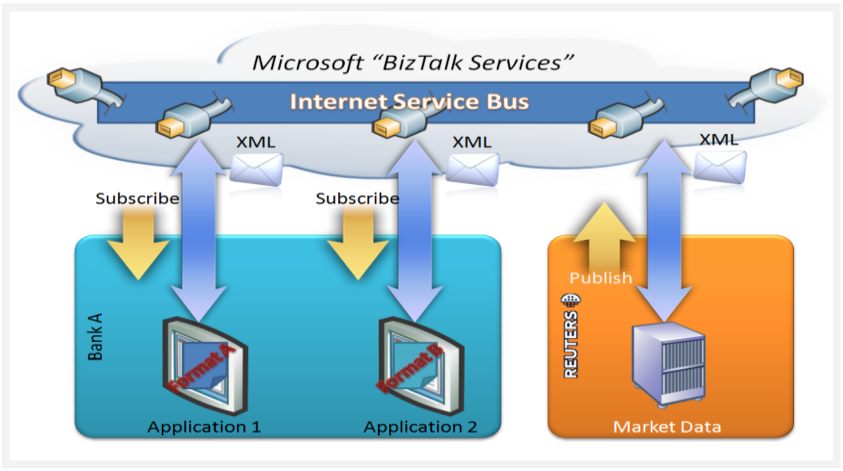


Figure 43: BizTalk Services Internet Service Bus [60]

The most important part of the *BizTalk Services ISB* is the supported relay binding. It allows easy access to other services that are secured behind a firewall. The automatic connect procedure is shown in Figure 44.

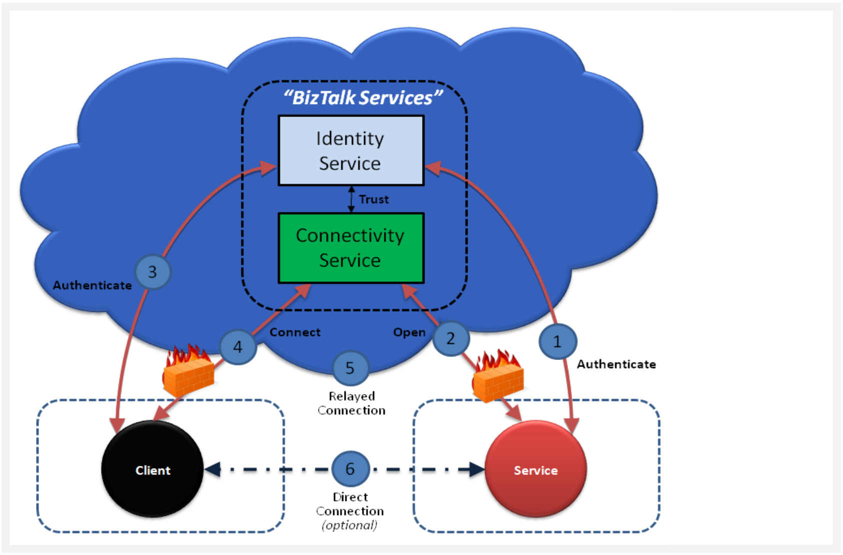


Figure 44: BizTalk Services – Relay Binding [60]

The *XcoBiztalkCommunicationService* contains the XVSM space implementation to connect space instances over the internet via the *BizTalk Services ISB*. For this purpose the *XcoBiztalkCommunicationService* inherits the *XcoWCFCommunicationService* and extends the *BizTalk* functionality. Here only the loading mechanism of the *BizTalk* specific settings from the configuration should be adapted. For a very simple solution a username token is used to authenticate the space service against the ISB. *WCF* covers all *BizTalk* details, and after correctly loading the new settings for the "normal" *WCF* communication service, it is ready to communicate across firewalls without changing anything on them. An example configuration can be found in Example 15.

```
<system.serviceModel>
<services><!-- For XVSM cores relayBinding for biztalk-->
  <service name="XcoSpaces.Kernel.Communication.WCF.SpacePortal"
         behaviorConfiguration="XCoreBehavior">
     <endpoint name="XVSMRelayEndpoint"
       contract="XcoSpaces.Kernel.Communication.WCF.IRemoteSpaceService"
       binding="relayBinding" bindingConfiguration="XVSMBTBinding"
       address="" />
  </services>
<client> <!-- For XVSM core clients relayBinding for biztalk -->
   <endpoint name="XVSMClientRelayEndpoint"
      contract="XcoSpaces.Kernel.Communication.WCF.IRemoteSpaceService"
      binding="relayBinding"
      bindingConfiguration="XVSMBTBinding"
      address="http://AddressToBeReplacedInCode/" />
</client>
<behaviors>
  <serviceBehaviors>
    <behavior  name="XCoreBehavior">
      <serviceThrottling maxConcurrentSessions="10000" />
    </behavior>
  </serviceBehaviors>
</behaviors>
<bindings>
  <relayBinding>
    <binding name="XVSMBTBinding"  maxReceivedMessageSize="512000"
      maxBufferSize="512000">
       <readerQuotas maxArrayLength="512000"   />
    </binding>
  </relayBinding>
</bindings>
</system.serviceModel>
```

Example 15: Configuration example for *BizTalk* services

The information and the experience that was collected with the *BizTalk Services* are involved in Microsoft's new *Windows Azure* platform [50] where this part is called *.NET* Services. The Azure platform is now also in *Community Technology Preview* (CTP) state for evaluation through January 2010.

6.3.4 Jabber / XMPP

Another solution to communicating in a barrier-free way over the internet can be established with the *Extensible Messaging and Presence Protocol* (XMPP) [51]. *XMPP* is an open XML-based protocol which is also known as *Jabber* [52]. The network of *XMPP* is managed decentralized similar to email, because anyone can run its own *XMPP* servers and no central master is needed. The *XMPP* server allows communicating over firewalls between different partners. Many clients implement an instant messenger, but over this protocol a simple application-to-application communication can also be established. An *XMPP* communication service is planned for the next version of the XVSM space.

6.4 Serialization

Serialization is the conversion process when an object is transformed to another representation for persisting on a storage medium or, in the main case of the space, for transferring data from one core instance to another.

6.4.1 *.NET* Serialization

The default serializer of the XVSM *.NET* space is the *WCF NetDataContractSerializer* [53]. It serializes and deserializes an instance of a type into XML stream or document using the supplied *.NET* Framework types. The *SerializationHelper* class can be found in the *WCF* communication service and implements the *ISerializationHelper* interface.

When a class must be serializable via *NetDataContractSerializer,* it must be marked with the *DataContractAttribute* or *SerializableAttribute*. For a class the *DataContractAttribute* helps to define which members of a class should be serialized and which should not. This is shown in Example 16. Here the class *Person* will be serialized together with its three properties with the *DataMember* attribute. On the other side when the *SerializableAttribute* (“*[Serializable]*”) is defined for a class, all *public* and *private* fields will be serialized.

```
[DataContract(Name = "Customer")]
public class Person
{
    [DataMember]
    public string FirstName;
    [DataMember]
    public string LastName;
    [DataMember]
    public int ID;

    public DateTime CreationDate;

    public Person(int ID, string FirstName, string LastName)
    {
        this.ID = ID;
        this.FirstName = FirstName;
        this.LastName = LastName;
        CreationDate = DateTime.Now;
    }
}
```
Example 16: DataContractAttribute for serialization with the NetDataContractSerializer

During performance testing of the different supported serializers of the *WCF*, the *NetDataContractSerializer* has the highest throughput with the best usability to use both serializing attribute definitions.

6.4.2 Interoperable Serialization – XVSM Protocol

An important feature in the XVSM space is the interoperability between XVSM nodes in a heterogeneous network. For this purpose an extensible protocol was developed to cover all functionalities and also to be prepared for new requirements in the future. More details about the *XML* protocol can be found in Chapter 7.

In the beginning we implemented the *XML* protocol without a schema definition and defined every single tag manually. In the next evaluation phase we decided to use an *XSD* (XML schema definition) with which validation checks can automatically be done for the basic structure and other details in an *XML* block. Since the *XSD* implementation is finished, a clean and clear structure exists and simple errors like typos are can be discovered and corrected much more quickly.

6.4.2.1 Processing

During the whole serialization process (shown in Figure 45), automatic *XML* generation tools were used to create a very small memory footprint and a high throughput. The serializer in the XVSM space is named *XMLSerializationHelper* and implements the *ISerializationHelper* interface members. The main functionalities such as serializing and deserializing are done with the corresponding *XcoXMLWriter* and *XcoXMLReader* classes.

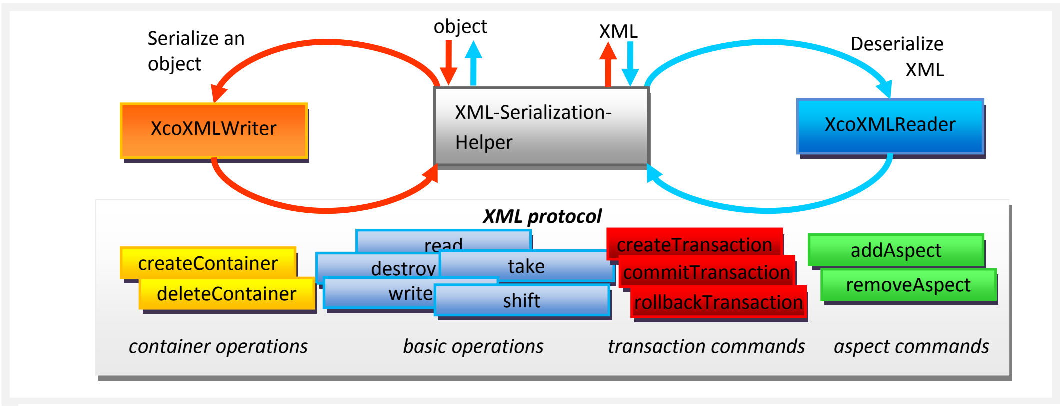


Figure 45: Overview XML serialization

In the *convert* classes every single core API command has its own set of rules for a correct conversion. If an error occurs during the convert process an *XcoSerializationException* will be thrown with further details.

7 XVSM - XML Protocol

Over the *XML* protocol, different implementations of the XVSM become able to communicate with each other. This makes it possible for example for *.NET* space (*XcoSpaces* [15]) to communicate with a *JAVA* space (*Mozartspaces* [13]) implementation and vice versa. Severin Ecker [54] built the initial version for this protocol. This version was then customized and extended for newly arisen needs. In the next subchapters an overview of the new protocol is given. In the current version the XML representations contains definitions for all core API (short CAPI) commands. The functionalities from the CAPI are very similar to the low level developer API described in Chapter 7.

7.1 Basic Elements

7.1.1 Values

In the value definition all variables are represented. For this the current protocol version supports the following *.NET* data types: *bool, byte, int, long, float, double, string, DateTime, byte[]* and *Uri*. In the current version complex objects cannot be automatically serialized. For the *XML* protocol complex objects must be converted to a tuple representation. This functionality is provided in the *Object-Tuple converter* which is a specialized serializer developed by Alexander Marek [55]. Over this the conversation between object and tuple representation is supported, and vice versa.

```
<value>
  <string>simple string value</string>
</value>
```

Example 17: XML protocol – values

When combination of data types is needed as a single object, a tuple representation must be created, which is shown in the following Chapter 7.1.2. This tuple definition can also be stored in a value instance.

7.1.2 Tuples

The tuple representation defines a collection of different values. This can be used, for example, to represent an entry in a *linda* coordinated container or the tuple can be added to a normal entry. In the Example 18 a tuple with three elements is shown on the left side. On the other side a template tuple is shown. Here only the field to be matched exactly contains specific values between the data type definitions; for all others only the data type is defined and no specific value is given (using the <null> element).

```
<tuple size="3">                      <tuple size="3">
  <value position="0">                  <value position="0">
    <integer>1</integer>                  <integer>1</integer>
  </value>                              </value>
  <value position="1">                  <value position="1">
    <string>test</string>                 <null>string</null>
  </value>                              </value>
  <value position="2">                  <value position="2">
    <long>10000</long>                    <null>long</null>
  </value>                              </value>
</tuple>                              </tuple>
```

Example 18: XML protocol – left: tuple with values, right: template tuple

7.1.3 Entries and Properties

These are the important elements in the *XML* protocol for simple data. On the one hand side, entries are responsible for defining the data in the containers, and on the other hand side, the properties manage the settings for the different command elements.

Everything that may be modified in a container must be an *entry*. In an entry a value instance can be held. When more than one entry is to be processed, a list of entries will be used with the XML tag entries.

For dynamic settings a single property can be defined over a keyword and value combination. The *XML* elements *addAspect*, *operational context*, *ipoints*, *selectors* and *coordinators* can specify a list of properties.

```
<entries>                             <properties>
  <entry>                               <property key="Name">
    <value>                               <value type="string">MasterKey</value>
      <string>test value 1</string>     </property>
    </value>                            <property key="Type">
  </entry>                                <value type="string">String</value>
  <entry>                               </property>
    <value>                             <property key="Value">
      <string>test value 2</string>       <value type="string">X1050</value>
    </value>                            </property>
  </entry>                            </properties>
</entries>
```
Example 19: XML protocol – left: entries, right: properties

7.1.4 Selectors and Coordinators

The selectors and coordinator tags are used for specifying the coordination details for all basic operations.

The *coordinator* tag is used to define the supported coordinators on a container. For a coordinator definition the tag *name* must be set. Using this tag, the corresponding coordinator will be selected. Additional properties can be defined for a coordinator as shown in the next example (Example 20).

The selector details are used to set the coordination information for basic operations. They are very similar to the coordinator definition. Additionally, the entry count that is to be read must be specified for reading operations (when all available entries are to be read, the count must be set to "-1").

```
<coordinators>                                 <selectors>
  <coordinator name="FifoCoordinator" />         <selector class="KeySelector" count="1">
  <coordinator name="KeyCoordinator">              <properties>
    <properties>                                     <property key="Name">
      <property key="Name">                            <value type="string">k</value>
        <value>                                      </property>
          <string>k</string>                         <property key="Type">
        </value>                                       <value type="string">String</value>
      </property>                                    </property>
      <property key="Type">                          <property key="Value">
        <value>                                        <value type="string">k4</value>
          <string>String</string>                    </property>
        </value>                                   </properties>
      </property>                                </selector>
    </properties>                              </selectors>
  </coordinator>
</coordinators>
```
Example 20: XML protocol – left: coordinators, right: selectors

7.2 Basic Protocol Structure

7.2.1 Command behavior

The communication protocol is based on asynchronous calls which will be handled by containers. When a request is generated for another space instance it must be specified which container the results are to be stored in, and when these results are available. The result can be accessed from this container aka "answer container". The answer container can also be a virtual container that is managed by the remote communication. The remote core instance that processes the request writes the result to the given answer container. With this mechanism it is possible to distribute a command from core *A* to *B* and receive the result at *C*.

In the XML every command starts with *capi* (*core API*). The root element has two simple properties. The first property *source* reveals the sending core instance and the second one *answerToContainerRef* defines where the result must be returned.

```
<capi source="tcpxml://coreAddrA" answerToContainerRef="tcpxml://
{coreAddrA}/containers/00000000-0000-0000-0200-000000000000">
…
</capi>
```

Example 21: XML protocol – general structure

For every single command an operational context can be used to add extended information not directly needed in the command itself. This can be used, for example, for a security mechanism to provide a username and password to ascertain if the request should be processed or not.

```
<capi <!-- capi parameters --> >
  <!-- operation -->
   …
  <operationContext>
    <properties>
      <property key="user">
        <value type="string">bob</value>
      </property>
      <property key="passHash">
        <value type="string">o5§gbI6oWf</value>
      </property>
    </properties>
  </operationContext>
</capi>
```

Example 22: XML protocol – operational context

7.2.2 Request / Response

For every request command supported, a corresponding response definition exists in the *XSD*. A short section of the *XSD* is shown in Example 23, where the request and response definitions for the create container operation are listed.

For the create container operation the available fields are defined. With that information a new container can be created. In the response as base information the originator message identity *requestId* is defined so that the correlating request can be found. Additionally a status flag is defined, which will be either "ok" or "error". When an error state is given an exception information is provided in the base response definition. The exceptions declaration includes an enumeration definition of all supported exceptions via the *name* property and text property for exception details. If the request was successfully processed, extended information is available. In the current case the resulting container reference of the new container is provided.

```
<!-- create container definition -->
<xsd:complexType name="CreateContainerType">
  <xsd:sequence>
    <xsd:element name="coordinators" type="CoordinatorListType"/>
  </xsd:sequence>
  <xsd:attribute name="transactionRef" type="xsd:anyURI"/>
  <xsd:attribute name="size"  type="xsd:int"/>
</xsd:complexType>

<!-- basic response definition -->
<xsd:complexType name="ResponseBaseType">
  <xsd:sequence>
    <xsd:element name="exception" type="ExceptionType" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="requestId" type="xsd:string" use="optional"/>
  <xsd:attribute name="status" type="ResponseStates" use="required"/>
</xsd:complexType>

<!—create container response definition -->
<xsd:complexType name="CreateContainerResponseType">
  <xsd:complexContent>
    <xsd:extension base="ResponseBaseType">
      <xsd:attribute name="containerRef" type="xsd:anyURI"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```
Example 23: XML protocol – XSD definition example for request and response of container create

7.3 Operations

7.3.1 Container operations

7.3.1.1 Create container

In the create container element all supported coordinator information can be defined. There is also the option to set a transaction reference to process the *container create* in an explicit transaction, otherwise an implicit transaction is used. The last parameter *size* specifies the maximal entry count in a container ("-1" represents infinite count).

```
<!-- create a container with linda coordination -->

<capi source="tcpxml://{coreAddrA}"
answerToContainerRef="tcpxml://{coreAddrA}/containers/00000000-0000-0000-2800-000000000000">
  <createContainer transactionRef="tcpxml://{coreAddrB}/transactions/77aace52-0000-0000-2700-
000000000000" size="-1">
    <coordinators>
      <coordinator name="LindaCoordinator">
        <properties />
      </coordinator>
    </coordinators>
  </createContainer>
</capi>
<!-- container successfully created -->
<capi source="tcpxml://{coreAddrB}"
answerToContainerRef="tcpxml://{coreAddrB}/containers/00000000-0000-0000-2800-000000000000">
  <write containerRef="tcpxml://{coreAddrA}/containers/00000000-0000-0000-2800-000000000000">
    <entries>
      <entry>
        <value>
          <response>
            <createContainer requestId="00000000-0000-0000-2800-000000000000" status="ok"
containerRef="tcpxml://{coreAddrB}/containers/4b0cac83-2169-4218-8469-fd934552fa8e" />
          </response>
        </value>
      </entry>
    </entries>
  </write>
</capi>
```
Example 24: XML protocol – create container

In the following operation explanations the response type will not especially be mentioned, this would go beyond the scope of this master thesis. Additionally every *GUID* in the following examples is replaced through the placeholder *GUID* for better readability.

7.3.1.2 Destroy container

To delete a container, only its container reference is required. A transaction reference and a timeout can be specified additionally.

```
<deleteContainer containerRef="tcpxml://{coreAddrB}/containers/{CRef-GUID}"
transactionRef="tcpxml://{coreAddrB}/transactions/{TRef-GUID }" />
```

Example 25: XML protocol – destroy container

7.3.2 Basic operations

The basic operations are divided into *read* and *write* operations. *Read*, *take* and *destroy* belong to the read operations, *write* and *shift* to the write operations.

7.3.2.1 Read operations

For the read operations the entries to be processed can be specified via the selector definition (see Chapter 7.1.4). The destination container can be defined by the container reference. The last property to be set is the timeout value. Furthermore a transaction reference can be specified for explicit transactions.

In the following xml fragment (see Example 26) a *read* command is shown that reads all entries in *fifo* order from the given container. When the other commands like *take* and *destroy* are used, the tag will simply be replaced.

```
<read containerRef="tcpxml://{coreAddrB}/containers/{CRef-GUID}" timeout="1000">
  <selectors>
    <selector name="FifoSelector" count="-1" />
  </selectors>
</read>
```

Example 26: XML protocol – example for a basic read operation

7.3.2.2 Write operations

The *write* operations are very similar to the *read* command. The main difference is that the entries with optional selector information which are to be written will be specified instead of only the selector definition.

```
<write containerRef="tcpxml://{coreAddrB}/containers/{CRef-GUID}" timeout="0">
    <entries>
      <entry>
        <value>
          <string>c</string>
        </value>
        <selectors>
          <selector name="LabelSelector" count="1">
            <properties>
              …
            </properties>
          </selector>
        </selectors>
      </entry>
      <entry>
          …
      </entry>
    </entries>
  </write>
</capi>
```

Example 27: XML protocol – example for a basic write operation

7.3.3 Transaction operations

7.3.3.1 Create transaction

The only parameter for a transaction create operation is the timeout.

```
<createTransaction timeout="-1" />
```

Example 28: XML protocol – transaction create

7.3.3.2 Transaction commit / rollback

For *commit* and *rollback* of a transaction only the transaction reference is needed.

```
<commitTransaction transaction="tcpxml://{coreAddrB}/transactions/{TRef-GUID}" />

<rollbackTransaction transaction="tcpxml://{coreAddrB}/transactions/{TRef-GUID}" />
```

Example 29: XML protocol – transaction commit / rollback

7.3.4 Aspect operations

7.3.4.1 Add aspect

The definitions for the aspect operations are a little more complicated than for the other commands. First the name must be specified. Using this parameter the corresponding registered implementation will be found. The registration can be done over the *microkernel* in the space. The *microkernel* concept allows dynamic bindings between contracts and current implementations, for more information see [21]. The next parameter *type* specifies on which platforms the aspect is hosted (*java*, *dotNet* and *interopt* are available). The *interopt* parameter is planned for scripting support in the next version of the XVSM space.

For the aspect the target type must be specified. Here the two types *space* and *container* can be selected. Depending on the target type a container reference must be specified if the aspect is a container aspect; space aspects do not need this definition. For clear specification of every single possible insertion point (short *ipoint*) an element in the enumeration collection exists. From the definition the space knows where the aspect must be registered. For a complete definition of aspect implementation specific information can be added over properties.

```
<addAspect name="NotificationAspect" type="dotNet" target="container"
containerRef="tcpxml://{coreAddrB}/containers/{CRef-GUID}">
  <ipoints>
    <ipoint>PostWrite</ipoint>
    <ipoint>PostContainerDestroy</ipoint>
    <ipoint>PostAddAspect</ipoint>
  </ipoints>
  <properties>
    <property key="cref">
      <value>
        <uri>tcpxml://{coreAddrB}/containers/{CRef-GUID}</uri>
      </value>
    </property>
    <property key="ncref">
      <value>
        <uri>tcpxml://{coreAddrB}/containers/{CRef-GUID}</uri>
      </value>
    </property>
  </properties>
</addAspect>
```

Example 30: XML protocol – add aspect

7.3.4.2 Remove aspect

Removing an aspect from the space is very similar to the adding procedure. Here the name, type and target definition are replaced by a single aspect reference. All other information is defined as in the add command, the only difference being that the set *ipoints* will be deregistered on the space.

```xml
<removeAspect aspectRef="tcpxml://{coreAddrB}/aspects/{ARef-GUID}" target="container"
containerRef="tcpxml://{coreAddrB}/containers/{CRef-GUID}">
  <ipoints>
    <ipoint>PostWrite</ipoint>
  </ipoints>
</removeAspect>
```

Example 31: XML protocol – remove aspect

8 The Low – Level API

The low level API provides access to all functionalities in the XVSM space. This means that on this level very large amount of parameters and overloaded methods exist to cover all possibilities. For this purpose the method versions with the most parameters will be explained. The low level API is not object oriented. A higher level API exists which is developed by Ralf Westphal.

8.1 XcoKernel

The *XcoKernel* that can be found in the namespace *XcoSpaces.Kernel* is the class with all user methods that will be shown in detail in the next sub chapters. The XcoKernel can be seen as the implementation of the XCore model. In the creating procedure of a new *XcoKernel* instance, all configuration settings will be loaded over the *microkernel*. The *microkernel* allows loading dynamic component bindings at runtime, for more details see [21].

```
public XcoKernel()
```

Example 32: low level API – Constructor of the XcoKernel

8.2 Operations

8.2.1 Container operations

8.2.1.1 Create container

A new container can be created using the create container methods. For correct specification, the maximal amount of entries in a container and the supported coordination types must be defined. All other parameters are for additional requirements like using a transaction or adding an operational context for an aspect.

```
public ContainerReference CreateContainer(String address, int size, params Selector[]
coordinationTypes)

public ContainerReference CreateContainer(String address, TransactionReference tref, int
size, params Selector[] coordinationTypes)

public ContainerReference CreateContainer(String address, TransactionReference tref, int
size, Selector[] coordinationTypes, OperationContext specificOpContext)

public ContainerReference CreateContainer(String address, TransactionReference tref, Guid id,
int size, params Selector[] coordinationTypes)

public ContainerReference CreateContainer(String address, TransactionReference tref, Guid id,
int size, params Selector[] coordinationTypes)

public ContainerReference CreateContainer(String address, TransactionReference tref, Guid id,
int size, Selector[] coordinationTypes, OperationContext specificOpContext)
```

Example 33: API - create container

CreateContainer	
address	The address of the kernel in format ip:port where the container must be created (null for local containers)
size	The maximum size of the container. Use -1 for an unbounded container.
coordinationTypes	The list of the coordination types for this container (fifo, lifo, vector, key, ...). A container must have at least one coordination type.
id	The id of the container. An exception will be thrown if a container with this id already exists. Using *Guid.Empty* automatically leads to the creation of a new id
tref	Reference to the transaction in which the operation is to be performed, or null if no transaction is to be used.

specificOpContext	The operation context that must be used specifically within this operation (can provide additional information for aspects).

8.2.1.2 Get container properties

The *GetProperty* methods give access to container and coordinator properties. All *GetProperty* methods that use the parameter *propertyName* are used to read information from coordinators, the others are for containers. Container properties will be read via the corresponding meta container. Only some special properties such as the entry count in the container will be directly processed. More information about meta containers can be found in [21].

With the *SetProperty* methods custom meta information can be written to the container.

```
public object GetProperty(ContainerReference cref, TransactionReference tref, int timeout,
String propertyName)

public object GetProperty(ContainerReference cref, TransactionReference tref, int timeout,
Selector selector, String propertyName)

public object GetProperty(ContainerReference cref, TransactionReference tref, int timeout,
Selector selector, String propertyName, OperationContext specificOpContext)

public object GetProperty(ContainerReference cref, TransactionReference tref, int timeout,
ContainerProperty prop)

public object GetProperty(ContainerReference cref, TransactionReference tref, int timeout,
ContainerProperty prop, OperationContext specificOpContext)

public void SetProperty(ContainerReference cref, TransactionReference tref, int timeout,
String propertyName, object propertyValue)

public void SetProperty(ContainerReference cref, TransactionReference tref, int timeout,
String propertyName, object propertyValue, OperationContext specificOpContext)
```

Example 34: API – publishing containers

PublishContainer / UnpublishContainer	
cref	Reference to the container where the property should be read/written
tref	Reference to the transaction in which the operation should be performed, or null if no transaction should be used
timeout	The timeout of the operation in milliseconds. Use *System.Threading.Timeout.Infinite* for infinite timeout.
propertyName	The name of the property that should be read/written.
prop	The container property that should be read
selector	The selector that defines from which coordinator of the container the property should be read
propertyValue	The property value that should be written
specificOpContext	The operation context that should be used specifically within this operation (can provide additional information for aspects).

8.2.1.3 Publishing container

With *PublishContainer* a container can be published in the lookup container which has got a static address. Every space instance has its own lookup container. In the lookup container other containers can be found via a unique name.

UnpublishContainer is the opposite to *PublishContainer*. Here a published container is removed from the lookup container.

```
public void PublishContainer(ContainerReference cref, TransactionReference tref, String name)

public void PublishContainer(ContainerReference cref, TransactionReference tref, String name,
OperationContext specificOpContext)

public ContainerReference UnpublishContainer(String address, TransactionReference tref,
String name)
```

```
public ContainerReference UnpublishContainer(String address, TransactionReference tref,
String name, OperationContext specificOpContext)
```
Example 35: API – publishing containers

PublishContainer / UnpublishContainer	
cref	The reference to the container to be published
name	The name of the published container
address	The address of the space where the container is published, in the form ip:port (null for local containers)
tref	Reference to the transaction in which the operation should be performed, or null if no transaction should be used.
specificOpContext	The operation context that should be used specifically within this operation (can provide additional information for aspects).

8.2.1.4 *Destroying a container*

The command *DestroyContainer* removes a container with all its entries from the space.

```
public void DestroyContainer(ContainerReference cref)

public void DestroyContainer(ContainerReference cref, TransactionReference tref, int timeout)

public void DestroyContainer(ContainerReference cref, TransactionReference tref, int timeout,
OperationContext specificOpContext)
```

Example 36: API – destroy containers

PublishContainer / UnpublishContainer	
cref	The reference to the container that should be destroyed
tref	Reference to the transaction in which the operation should be performed, or null if no transaction should be used.
timeout	The timeout of the operation in milliseconds. Use *System.Threading.Timeout.Infinite* for infinite timeout.
specificOpContext	The operation context that should be used specifically within this operation (can provide additional information for aspects).

8.2.1.5 *Named container*

With the named containers method, the user is given access to an automatic container publishing mechanism. During create the container will automatically be published, and is un-published on destroy. With the *GetNamedContainer* an easy access to a published container will be supported.

```
public ContainerReference CreateNamedContainer(String address, String name, int size, params
Selector[] coordinationTypes)

public ContainerReference CreateNamedContainer(String address, TransactionReference tref,
String name, int size, params Selector[] coordinationTypes)

public ContainerReference CreateNamedContainer(String address, TransactionReference tref,
String name, int size, Selector[] coordinationTypes, OperationContext specificOpContext)

public ContainerReference GetNamedContainer(String address, String name)

public ContainerReference GetNamedContainer(String address, String name, OperationContext
specificOpContext)

public void DestroyNamedContainer(String address, String name)

public void DestroyNamedContainer(String address, TransactionReference tref, int timeout,
String name)

public void DestroyNamedContainer(String address, TransactionReference tref, int timeout,
String name, OperationContext specificOpContext)
```
Example 37: API - named container

CreateNamedContainer	
address	The address of the kernel in format ip:port where the container should be created. (null for local containers)
name	The name of the published Container
size	The maximum size of the container. Use -1 for an unbounded container.
coordinationTypes	The list of the coordination types for this container (fifo, lifo, vector, key, ...). A container must have at least one coordination type.
id	The id of the container. An exception will be thrown if a container with this id already exists. Using *Guid.Empty* automatically leads to the creation of a new id
tref	Reference to the transaction in which the operation should be performed, or null if no transaction should be used.
specificOpContext	The operation context that should be used specifically within this operation (can provide additional information for aspects).

8.2.2 Basic operations

The basic operations can be split up to two groups: read and write operations. A detailed description about the behavior of every basic operation is shown in Chapter 4.4.1.

8.2.2.1 Read operations

The normal *read* operation only reads entries from the space. With the *take* operation the read entries will be additionally removed from the space. The last read operation is *destroy*, which works in a similar way to *take*, but does not give the read entries back to the user.

```
public List<IEntry> Read(ContainerReference cref, TransactionReference tref, int timeout,
params Selector[] selectors)

public List<IEntry> Read(ContainerReference cref, TransactionReference tref, int timeout,
Selector[] selectors, OperationContext specificOpContext)

public List<IEntry> Take(ContainerReference cref, TransactionReference tref, int timeout,
params Selector[] selectors)

public List<IEntry> Take(ContainerReference cref, TransactionReference tref, int timeout,
Selector[] selectors, OperationContext specificOpContext)

public void Destroy(ContainerReference cref, TransactionReference tref, int timeout, params
Selector[] selectors)

public void Destroy(ContainerReference cref, TransactionReference tref, int timeout,
Selector[] selectors, OperationContext specificOpContext)
```

Example 38: API – read operations: read, take and destroy

Read / Take / Destroy	
cref	Reference to the container where the entries should be read/taken/destroyed
tref	Reference to the transaction in which the operation should be performed, or null if no transaction should be used.
timeout	The timeout of the operation in milliseconds. Use *System.Threading.Timeout.Infinite* for infinite timeout.
selectors	The list of selectors by which the entries are selected from the container
specificOpContext	The operation context that should be used specifically within this operation (can provide additional information for aspects).

8.2.2.2 Write operations

In the *write* operations new entries can be added to the space. A *write* operation will only be processed when all conditions are fulfilled. *Shift* in contrast, overwrites existing entries if it is needed to add the new entries to the container.

```
public void Write(ContainerReference cref, TransactionReference tref, int timeout, params
IEntry[] entries)

public void Write(ContainerReference cref, TransactionReference tref, int timeout, IEntry[]
entries, OperationContext specificOpContext)

public void Shift(ContainerReference cref, TransactionReference tref, int timeout, params
IEntry[] entries)

public void Shift(ContainerReference cref, TransactionReference tref, int timeout, IEntry[]
entries, OperationContext specificOpContext)
```
Example 39: API – write operations: write and shift

Write / Shift	
cref	Reference to the container where the entries should be written/shifted
tref	Reference to the transaction in which the operation should be performed, or null if no transaction should be used.
timeout	The timeout of the operation in milliseconds. Use *System.Threading.Timeout.Infinite* for infinite timeout.
entries	The entries that should be written/shifted
specificOpContext	The operation context that should be used specifically within this operation (can provide additional information for aspects).

8.2.3 Transaction operations

Using these methods all transaction states can be managed. With create, a new transaction instance can be created. The *prepare* state guarantees that a successful *commit* can be achieved. Commit stores all changes of the transaction in the space and releases all locks. During rollback all changes of the transaction will be reverted to the original state. For more details about transactions see Chapter 5.

```
public TransactionReference CreateTransaction(String address, int timeout)

public TransactionReference CreateTransaction(String address, int timeout, OperationContext
specificOpContext)

public void PrepareTransaction(TransactionReference tref)

public void PrepareTransaction(TransactionReference tref, OperationContext specificOpContext)

public void CommitTransaction(TransactionReference tref)

public void CommitTransaction(TransactionReference tref, OperationContext specificOpContext)

public void RollbackTransaction(TransactionReference tref)

public void RollbackTransaction(TransactionReference tref, OperationContext
specificOpContext)
```
Example 40: API – transaction management

Create- / Prepare- / Commit- / Rollback- Transaction	
address	The address of the kernel in format ip:port where the container should be created. (null for local containers)
tref	Reference to the transaction in which the operation should be performed
timeout	The timeout of the operation in milliseconds. Use *System.Threading.Timeout.Infinite* for infinite timeout.
specificOpContext	The operation context that should be used specifically within this operation (can provide additional information for aspects).

8.2.4 Aspects

With aspects the behavior of the space can be customized. For more information about aspects see [21]. In the next chapters the two main groups of aspects will be explained, first the container aspects then the space variant.

8.2.4.1 Add Container Aspects

Here two different creation types (local and the remote) are supported. With the local variant the container aspect can be directly referenced. When remote instantiation is used, the given type (*aspectType*) or name (*aspectName*) must be already registered in the space. The given properties will be automatically injected into the aspect instances that are using the *XcoSpaces.Kernel.Microkernel.PropertyAttribute* attribute.

```
public AspectReference AddContainerAspect(ContainerReference cref, ContainerAspect aspect)

public AspectReference AddContainerAspect(ContainerReference cref, ContainerAspect aspect,
OperationContext specificOpContext)

public AspectReference AddContainerAspect(ContainerReference cref, ContainerAspect aspect,
params ContainerIPoint[] iPoints)

public AspectReference AddContainerAspect(ContainerReference cref, ContainerAspect aspect,
ContainerIPoint[] iPoints, OperationContext specificOpContext)

public AspectReference AddContainerAspect(ContainerReference cref, Type aspectType,
Dictionary<string, object> properties)

public AspectReference AddContainerAspect(ContainerReference cref, Type aspectType,
Dictionary<string, object> properties, params ContainerIPoint[] iPoints)

public AspectReference AddContainerAspect(ContainerReference cref, Type aspectType,
Dictionary<string, object> properties, OperationContext specificOpContext)

public AspectReference AddContainerAspect(ContainerReference cref, Type aspectType,
Dictionary<string, object> properties, ContainerIPoint[] iPoints, OperationContext
specificOpContext)

public AspectReference AddContainerAspect(ContainerReference cref, string aspectName,
Dictionary<string, object> properties, ContainerIPoint[] iPoints, OperationContext
specificOpContext)

public AspectReference AddContainerAspect(ContainerReference cref, string aspectName,
Dictionary<string, object> properties, params ContainerIPoint[] iPoints)
```

Example 41: API – create container aspects

AddContainerAspect	
cref	Reference to the Container where the Aspect should be added
aspect	The aspect that should be added
aspectType	The type of the aspect that should be added
aspectName	The predefined name of the aspect that should be added
iPoints	The insertion points where the Aspect should be added in the Container
properties	The list of properties for instantiating the aspect
specificOpContext	The operation context that should be used specifically within this operation (can provide additional information for aspects).

8.2.4.2 Add Space Aspects

The create behavior of space aspects is very similar to the container aspects, with the main difference being that space aspects are valid for the whole space and not registered to certain containers.

```
public AspectReference AddSpaceAspect(SpaceAspect aspect)

public AspectReference AddSpaceAspect(SpaceAspect aspect, params SpaceIPoint[] iPoints)

public AspectReference AddSpaceAspect(SpaceAspect aspect, OperationContext specificOpContext)

public AspectReference AddSpaceAspect(SpaceAspect aspect, SpaceIPoint[] iPoints,
OperationContext specificOpContext)

public AspectReference AddSpaceAspect(string address, Type aspectType, Dictionary<string,
object> properties)

public AspectReference AddSpaceAspect(string address, Type aspectType, Dictionary<string,
object> properties, params SpaceIPoint[] iPoints)

public AspectReference AddSpaceAspect(string address, Type aspectType, Dictionary<string,
object> properties, OperationContext specificOpContext)

public AspectReference AddSpaceAspect(string address, Type aspectType, Dictionary<string,
object> properties, SpaceIPoint[] iPoints, OperationContext specificOpContext)

public AspectReference AddSpaceAspect(string address, string aspectName, Dictionary<string,
object> properties, params SpaceIPoint[] iPoints)

public AspectReference AddSpaceAspect(string address, string aspectName, Dictionary<string,
object> properties, SpaceIPoint[] iPoints, OperationContext specificOpContext)
```

Example 42: API – create space aspects

AddContainerAspect	
address	The address of the space where the aspect should be added
aspect	The aspect that should be added
aspectType	The type of the aspect that should be added
aspectName	The predefined name of the aspect that should be added
iPoints	The insertion points where the aspect should be added in the space
properties	The list of properties for instantiating the aspect
specificOpContext	The operation context that should be used specifically within this operation (can provide additional information for aspects).

8.2.4.3 Remove Aspects

All added aspects can also be removed from the space. The following code example shows all available methods for removing aspects.

```
public void RemoveAspect(AspectReference aref)

public void RemoveAspect(AspectReference aref, OperationContext specificOpContext)

public void RemoveContainerAspect(ContainerReference cref, AspectReference aref)

public void RemoveContainerAspect(ContainerReference cref, AspectReference aref,
OperationContext specificOpContext)

public void RemoveContainerAspect(ContainerReference cref, AspectReference aref, params
ContainerIPoint[] iPoints)

public void RemoveContainerAspect(ContainerReference cref, AspectReference aref,
ContainerIPoint[] iPoints, OperationContext specificOpContext)

public void RemoveSpaceAspect(AspectReference aref, params SpaceIPoint[] iPoints)

public void RemoveSpaceAspect(AspectReference aref, SpaceIPoint[] iPoints, OperationContext
specificOpContext)
```

Example 43: API – removing aspects

RemoveAspect / RemoveContainerAspect / RemoveSpaceAspect	
aref	Reference to the aspect that should be removed
cref	Reference to the container where the aspect should be removed
iPoints	The insertion points where the aspect should be removed
specificOpContext	The operation context that should be used specifically within this operation (can provide additional information for aspects).

8.2.5 Notifications

The automatic notification mechanism allows getting an event for every change that the user has registered on a container. The notification mechanism itself is space internal implemented with aspects. A notification can be easily removed from the space with the Stop method of the Notification class. When the underlying container is destroyed the notification will also be stopped.

```
public Notification CreateNotification(ContainerReference cref, TransactionReference tref,
bool onRead, bool onTake, bool onDestroy, bool onWrite, bool onShift)

public Notification CreateNotification(ContainerReference cref, TransactionReference tref,
bool onRead, bool onTake, bool onDestroy, bool onWrite, bool onShift, OperationContext
specificOpContext)
public Notification CreateReadNotification(ContainerReference cref, TransactionReference
tref, params ReadOperation[] operations)
public Notification CreateReadNotification(ContainerReference cref, TransactionReference
tref, ReadOperation[] operations, OperationContext specificOpContext)
public Notification CreateWriteNotification(ContainerReference cref, TransactionReference
tref, params WriteOperation[] operations)
public Notification CreateWriteNotification(ContainerReference cref, TransactionReference
tref, WriteOperation[] operations, OperationContext specificOpContext)
```
Example 44: API – create notification

CreateNotification	
cref	Reference to the container the notification should be created on
tref	Reference to a transaction, if the notification should run within a transaction, or null (in which case the notification will only notify about operations after they have been committed).
onRead	True if the notification should trigger on *read* operations
onTake	True if the notification should trigger on *take* operations
onDestroy	True if the notification should trigger on *destroy* operations
onWrite	True if the notification should trigger on *write* operations
onShift	True if the notification should trigger on *shift* operations
operations	List of read or write operations for which a Notification should be thrown
specificOpContext	The operation context that should be used specifically within this operation (can provide additional information for aspects).

8.2.6 Communication Management

With the method *Start* the communication service of the space will be started. If no communication service has been added or configured, an *XcoWCFCommunicationService* will be used by default. Before the communication is stopped, all connections can be closed carefully using the *ReleaseCommunicationConnections*. The method *Stop* closes all communication channels of the remote communication service.

The operation context can be used to provide information to aspects within an operation. Properties of the default operation context are added to every operation that is called in the *XcoKernel*.

The method *Close* shuts down the kernel and stops the server for remote communication if running. All communication currently operating within the kernel will result in an *XcoOperationTimeoutException*.

The *Dispose* method cleans up all resources by calling *Close*.

```
public void Start()
public void Start(string ipAddress, int port)
public void Start(int port)
public void Stop()
public void ReleaseCommunicationConnections()
public void AddService(IXcoService service)
public OperationContext DefaultOperationContext {get;}
public void Close()
public void Dispose()
```
Example 45: API – communication start /stop and management functions

Start	
ipAddress	The IP address on which the service should be running (null if the service should decide itself)
port	The local port on which the server should be running. If the port is 0, a random port number between 8000 and 9000 will be generated
service	Adds a service to the XcoKernel. Only services of type IXcoCommunicationService are supported by now (other services must be added at kernel instantiation

9 Future Work

The current XVSM implementation offers a solid base for a powerful space based middleware solution. Looking further ahead we may expect that the XVSM space will be able to demonstrate whether the extension mechanisms are ready for real world scenarios or not. For subsequent versions, some interesting features can be imagined.

Based on the experience from the space-based computing team the next step is to undertake an intensive code review before new features will be integrated into the space. In this context a very true statement was made by Ralf Westphal on the last review in a parallel project: “Even source code needs vacation”. This code review should be done by an external team to validate the architecture as the realized solution for the different tasks. Through this code review new ideas can be discussed and the quality of the whole project will be enhanced. During the review process new requirements or ideas can also be collected for new features. After the review and rework the code base should be ready for changes and new functionalities can also be added and integrated.

A very interesting part of the operation of the space is the scalability in the future. The already existing performance tests [56] should be expanded. On the one hand side it is very interesting to see how many requests the system can process without encountering any problems, and on the other hand side it is important to identify bottlenecks in the system. If such performance problems are detected, the problem needs to be identified and adequate solutions have to be found.

For subsequent versions additional communication services are planned. As mentioned before, an *XMPP* (*Jabber*) service should be one of the next communication services to enable barrier free communication over the internet. (For this purpose software development kits are already available for the *.NET* framework.) For better performance, a *Jabber* server can also be hosted which only handles requests from space instances.

Another important feature for more convenient interoperable communication will be an automatic mechanism that can convert complex objects to a *XML* representation. The current implementation can only manage primitive data types; for this purpose the tuple-object converter from Alex Marek can be used for converting on the *.NET* side. As a next step, an implementation is needed for the *JAVA* side to automatically convert tuples to *JAVA* objects. Then the interoperability functionality should be tested with complex objects between the two platforms.

In subsequent versions scripting languages like *IronPhyton* [57] should be integrated into the space for dynamically adding scripted aspects. Using these scripting languages will dramatically increase the possibilities for what the user can do with aspects. The security mechanisms must be upgraded relative to the usage of scripting languages and so more options to make changes in the space are available. Here special security functionalities are needed like script originator authorization and authentication, script access validation and many more. No unchecked script should be processed by the space.

A simple persistency profile exists in the current space that works with *db4o* [58] and was developed by Alexander Marek [59]. With an automatic persistency mode the space is able to arrive at the same state after finishing the start sequence as it was before the last shutdown. In one of the next versions the internal space support for persistency functionality should be enhanced, by persistency modules.

Another interesting feature for a future space version is an automatic lookup functionality. A basic lookup approach has been implemented in the high level API which was developed by Ralf Westphal. For subsequent versions the lookup mechanisms should be enhanced so that all currently running core instances can be found together via a decentralized naming service, making it possible to simply search for and find every registered container.

For maintenance and monitoring new tools should be developed. Here information should be available such as memory and processor consumption for every space instance, average number of processed messages, container count and so on. With these tools an administrator can obtain more details about the current state of the space. There should as well be an option to collect statistical information in the form of reports. Another monitoring approach is a development tool that is able to show the container structure and the containing data in the containers. This would be very useful for debugging and troubleshooting, since finding an error in a distributed system is not normally an easy task.

In summary, there are many possibilities for upgrading the code to add new functionalities; thus, a great amount of work is yet to be done.

10 Conclusion

XVSM (eXtensible Virtual Shared Memory) is a new middleware that allows collaboration of software components over shared data structures which are managed in containers. Every container can handle multiple coordination patterns which allow to automatically handle the data in the container. It is possible to simply add customized features in the current space and notifications enable sending of events when changes happen. The transaction support allows concurrent requests for more throughput in the space without consistency problems. The space is based on a *P2P* [9] infrastructure and allows the use of different transport protocols and also has its own interoperable protocol for communication between different implementations of the XVSM space.

Further, the implementation details of *XcoSpaces*, which is the *.NET* implementation of XVSM, were used to show the realization of different concepts in the XVSM model. The following topics such as coordination, transactions, locking and communication were shown in detail.

In the coordination chapter all currently available coordination patterns were explained in detail for all basic operations (read, take, shift, write and destroy). The data is managed in the container using coordinator and selector pairs. For more flexibility the concept allows new coordinator and selector pairs to be integrated into the space very easily.

For good performance concurrent requests are supported in the XVSM space. The transaction management and the included locking mechanism are responsible for ensuring that there are no inconsistencies in the data when two requests are made on the same data. For each request the user may specify an explicit transaction reference, otherwise an implicit transaction will be used. The locking mechanism supports container and entry locking: container locking only allows a single write operation at a time, whereas in entry locking multiple write operations can be carried out concurrently when the entries to be written are different. The coordinators on the containers decide which locking mode should be used.

The document also introduced the communication concept of the space, which allows various transport protocols to communicate with each other. The available *TCP*, *WCF*, *BizTalk* communication services are explained in detail in the message processing in the *XcoSpaces*. Different serialization mechanisms are supported for communication. A *XML* protocol is specified for interoperable communication across programming languages. This protocol enables the communication between *XcoSpaces* [15] and *MozartSpaces* [13] (the *JAVA* reference implementation of XVSM).

The API for developer was shown in detail. Here all methods for the management of the *XcoSpaces* were shown.

With the functionalities that are combined in the new middleware XVSM we hope to contribute our part for new and efficient technology to make the development of distribution applications easier. Through this alternative approach based on different coordination mechanisms and notifications the entire communication process will become more natural and save development resources and time.

11 References

1. **Internet World Stats.** World Internet Users and Population Stats. *Internet Usage Statistics.* [Online] Last visited: 2009-11-03. http://www.internetworldstats.com/stats.htm.

2. **Tim O'Reilly.** Design Patterns and Business Models for the Next Generation of Software. *What Is Web 2.0.* [Online] Last visited: 2009-10-06. http://oreilly.com/web2/archive/what-is-web-20.html.

3. **Sesum-Cavic, Vesna and Kühn, eva.** A Swarm Intelligence Appliance to the Construction of an Intelligent Peer-to-Peer Overlay Network. *1st Workshop on Coordination in Complex Software Intensive Systems (COCOSS-2010).* 2010, Co-located with CISIS 2010: 4th International Conference on Complex, Intelligent and Software Intensive Systems.

4. **Kühn, eva and Sesum-Cavic, Vesna.** *A Space-Based Generic Pattern for Self-Initiative Load Balancing Agents.* Utrecht University, The Netherlands : accepted for The 10th Annual International Workshop "Engineering Societies in the Agents' World" (ESAW 2009), 2009.

5. **Sesum-Cavic, Vesna and Kühn, eva.** *Peer-to-Peer Overlay Network based on Swarm Intelligence.* Utrecht University, The Netherlands : accepted as Poster Paper for The 10th Annual International Workshop "Engineering Societies in the Agents' World" (ESAW 2009), 2009.

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

7. **Kühn, eva; Mordinyi, Richard; Keszthelyi, László; Schreiber, Christian.** *Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems.* Budapest, Hungary : International Foundation for Autonomous Agents and Multiagent Systems, 2009.

8. **Kühn, eva, Mordinyi, Richard and Schreiber, Christian.** *An Extensible Space-based Coordination Approach for Modeling Complex Patterns in Large Systems for Analysing.* Porto Sani, Greece : Leveraging Applications of Formal Methods, Verification and Validation, 2008.

9. **Schollmeier, Rüdiger.** *A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications.* Los Alamitos, CA, USA : IEEE Computer Society, 2001.

10. **Gelernter, David.** *Generative communication in linda.* s.l. : ACM Trans. Program. Lang. Syst., 1985.

11. **Gelernter, David and Carriero, Nicholas.** *Coordination Languages and their Significance.* New York, NY, USA : Commun. ACM, 1992.

12. **Sun Microsystems.** Java . [Online] Sun Microsystems, Inc, Last visited: 2009-11-01. http://java.sun.com/.

13. **MozartSpaces.** [Online] Last visited: 2009-10-03. http://www.mozartspaces.org.

14. **Microsoft.** .Net Framework Developer Center. *MSDN.* [Online] Last visited: 2009-08-20. http://msdn2.microsoft.com/en-us/netframework/default.aspx.

15. **XCOORDINATION.** Software Technologies. [Online] Last visited: 2009-09-02. http://www.xcoordination.com/.

16. **Space based computing group.** [Online] Last visited: 2009-11-09. http://www.spacebasedcomputing.org.

17. **eva Kühn.** Institute of computer languages. [Online] Last visited: 2009-11-07. http://www.complang.tuwien.ac.at/eva.

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

19. **Kühn, eva and Nozicka, Georg.** *Post-Client/Server Coordination Tools.* s.l. : Coordination Technology for Collaborative Applications, Wolfram Cohen, Gustaf Neumann (eds.), Springer Series Lecture Notes in Computer Science, 1998.

20. **Kühn, eva.** *Virtual shared memory for distributed architectures.* Commack, NY, USA : Nova Science Publishers, Inc., 2001.

21. **Scheller, Thomas.** *Design and Implementation of XcoSpaces, the .Net Reference Implementation of XVSM: Core Architecture and Aspects.* TU-Vienna, Insititute of Computer Languages, SBC-Group : Master Thesis, 2008.

22. **Androutsellis-Theotokis, Stephanos and Spinellis, Diomidis.** *A survey of peer-to-peer content distribution technologies.* s.l. : ACM, 2004.

23. **GigaSpaces Technologies Inc.** *GigaSpaces.* [Online] Last visited: 2009-10-03. http://www.gigaspaces.com.

24. **Creswell, Dan.** *The Blitz Project.* [Online] Last visited: 2009-10-02. http://www.dancres.org/blitz.

25. **Freeman, Eric, Arnold, Ken and Hupfer, Susanne.** *JavaSpaces Principles, Patterns, and Practice.* UK : Addison-Wesley Longman Ltd., 1999.

26. **LighTS.** [Online] Last visited: 2009-10-06. http://lights.sourceforge.net.

27. **Kühn, eva and Schmied, Fabian.** *XL-AOF: lightweight aspects for space-based computing.* Grenoble, France : ACM, 2005.

28. **Kühn, E., Mordinyi, R., Keszthelyi, L., Schreiber, Bessler, S., and Tomic, S.** *Aspect-oriented Space Containers for Efficient Publish/Subscribe Scenarios in Intelligent Transportation Systems.* s.l. : The 11th International Symposium on Distributed Objects, Middleware, and Applications (DOA'09), 2009.

29. **Jini.** Jini.org. [Online] Last visited: 2009-11-04. http://www.jini.org.

30. **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.

31. **Craß, Stefan, Kühn, eva and Salzer, Gernot.** *Algebraic Foundation of a Data Model for an Extensible Space-Based Collaboration Protocol.* Calabria, Italy : Thirteenth International Database Engineering & Applications Symposium (IDEAS), 2009.

32. **Schmidt, Douglas C.; Stal, Michael; Rohnert, Hans; Buschmann, Frank.** *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects.* s.l. : John Wiley & Sons, 2000.

33. **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, 2008.

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

35. **Kemper, Alfons and Eickler, André.** *Datenbanksysteme - Eine Einführung.* München, Wien : R. Oldenbourg Verlag, 2001.

36. **Haerder, Theo and Reuter, Andreas.** *Principles of transaction-oriented database recovery.* New York, NY, USA : ACM Comput. Surv., 1983.

37. **Weikum, Gerhard and Vossen, Gottfried.** *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001.

38. **Coffman, E. G., Elphick, M. and Shoshani, A.** *System Deadlocks.* New York, NY, USA : ACM Computing Surveys, 1971.

39. **Tanenbaum, Andrew S. and Van Steen, Maarten.** *Distributed Systems: Principles and Paradigms .* USA : Prentice Hall, 2001.

40. **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, 2008.

41. **Microsoft.** Windows Communication Foundation. *MSDN.* [Online] Last visited: 2009-09-03. http://msdn2.microsoft.com/en-us/netframework/aa663324.aspx.

42. **Microsoft.** DCOM Architecture. *MSDN.* [Online] Last visited: 2009-09-05. http://msdn.microsoft.com/de-de/library/ms809311.aspx.

43. **Microsoft.** COM+ (Component Services). *MSDN.* [Online] Last visited: 2009-09-05. http://msdn.microsoft.com/en-us/library/ms685978.aspx.

44. **Microsoft.** Microsoft Message Queuing. *Microsoft Corporation.* [Online] Last visited: 2009-09-06. http://msdn.microsoft.com/en-us/library/ms711472.aspx.

45. **Microsoft.** Web Services Enhancements. *MSDN.* [Online] Last visited: 2009-09-05. http://msdn.microsoft.com/en-us/library/dd560722.aspx.

46. **Microsoft.** Introduction to Building Windows Communication Foundation Services. *MSDN.* [Online] Last visited: 2009-09-06. http://msdn.microsoft.com/en-us/library/aa480190.aspx.

47. **McMurtry, Craig; Mercuri, Marc; Watling, Nigel; Winkler, Matt.** *Windows Communication Foundation Unleashed (WCF) (Unleashed).* s.l. : Sams Publishing, 2007.

48. **Microsoft.** Windows Communcation Foundation Bindings. *MSDN.* [Online] Last visited: 2009-09-07. http://msdn.microsoft.com/en-us/library/ms733027.aspx.

49. **Microsoft.** Services. *Biztalk.Net.* [Online] Last visited: 2008-04-06. http://labs.biztalk.net.

50. **Microsoft.** Windows Azure Platform. *Windows Azure Platform.* [Online] Last visited: 2009-09-05. http://www.microsoft.com/windowsazure/.

51. **XMPP Standards Foundation.** XMPP Standards Foundation. [Online] Last visited: 2009-09-06. http://xmpp.org/.

52. **Jabber.** Jabber.org. [Online] Last visited: 2009-09-06. http://www.jabber.org.

53. **Microsoft.** .NET Framework - NetDataContractSerializer. *MSDN.* [Online] Last visited: 2009-09-05. http://msdn.microsoft.com/en-us/library/system.runtime.serialization.netdatacontractserializer.aspx.

54. **Ecker, Severin.** *Communication protocols in xvsm-design and implementation.* TU-Vienna, Insititute of Computer Languages, SBC-Group : Master Thesis, 2007.

55. **Marek, Alexander.** *Profile for XcoSpaces: TupleConverter, Praktikum.* Vienna University of Technology, Institute of Computer Languages, E185/1 : Space Based Computing Group, 2008.

56. **Kühn, eva; Mordinyi, Richard; Moritsch, Hans; Scheller, Thomas; Schreiber, Christian.** *A Staged-driven Architecture style for a Scalable Space-based Middleware.* TU-Vienna, Insititute of Computer Languages, SBC-Group : Technical Report, 2008.

57. **IronPython.** IronPython. [Online] Last visited: 2009-11-01. http://www.codeplex.com/wikipage?ProjectName=IronPython.

58. **Versant Corp.** db4o. *db4o - open source object database.* [Online] Last visited: 2009-11-02. http://www.db4o.com.

59. **Marek, Alexander.** *Profile for XcoSpaces: AdvancedPersistency, Praktikum.* Vienna University of Technology, Institute of Computer Languages, E185/1 : Space Based Computing Group, 2008.

60. **Weyer, Christian.** In the Cloud Connect Your Services with the Internet Service Bus. *MSDN.* [Online] Last visited: 2009-09-04. http://download.microsoft.com/download/3/a/3/3a31f2e5-39fe-4df0-ba40-a5fc96b14a05/A103_In the cloud_Services with the Internet Service Bus.pptx.