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

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

MASTERARBEIT

XVSM Tutorial and Application Scenarios

Ausgeführt am
Institut für Computersprachen
Abteilung für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter Anleitung von
Ao. Univ. Prof. Dipl.-Ing. Dr. eva Kühn

durch

Michael Wittmann
Spitzweg 43
A-1210 Wien
Matr.Nr.: 0026159

Wien, im März 2008 ________________________

Table of contents

Abstract (English) ... 3

Abstract (German) ... 4

Coding standard ... 5

How to maintain the tutorial ... 6

Application scenarios ... 9

Tutorial ... 40

Used Literature and References ... 99

Erklärung ... 102

Note that the printed document has another pagination and a slightly different Table of contents than this document. The reason is that in the printed version, the Application scenarios and the Tutorial are handled as extra documents, separated by a green page, which is not possible here. Thus also the text format may differ slightly.

Abstract (English)

XVSM (eXtensible Virtual Shared Memory) is middleware to let applications write data to a shared memory that can be read by other applications. The data is distributed over a set of peers, which can read, write, take and delete it, independent on which peer the data is really stored. If one peer fails, the data is not lost, as it is available also on other peers.

This master thesis should provide the necessary information to learn and understand the features and possibility of space-based computing by means of XVSM and its Java implementation, called MozartSpaces. It was shown to be usable for this task by handing it to about 80 students and incorporate their feedback to the work.
The application scenario document should give application designers an insight into the possibilities they have when using XVSM. The purpose of this master thesis is not to give a complete documentation of the current state of XVSM, but to arouse interesting in the reader to use XVSM and also to give him/her a base knowledge of its features, possibilities and its usage.

Abstract (German)

XVSM (eXtensible Virtual Shared Memory) ist Middleware, um es Applikationen zu ermöglichen, Daten aus einem gemeinsamen Datenspeicher zu lesen. Die Daten sind über mehrere Peers verteilt, welche die Daten lesen, schreiben, entnehmen oder löschen können, unabhängig davon, auf welchem Peer die Daten tatsächlich gespeichert sind. Wenn ein einzelner Peer ausfällt, sind die Daten nicht verloren, da sie auf mehreren Peers verteilt sind.

Diese Masterarbeit soll die notwendigen Informationen bieten, um die Features und Möglichkeiten von Space-Based Computing anhand von XVSM und seiner Java-Implementierung, genannt MozartSpaces, kennen zu lernen. Es wurde gezeigt, dass diese Aufgabe erfüllt werden konnte, indem es an etwa 80 Studenten bereits praktisch erprobt wurde und deren Feedback laufend in die Arbeit einfloss.
Das Application scenarios (Anwendungsszenarien-) Dokument soll Anwendungsdesignern einen Einblick in die Möglichkeiten bieten, wenn sie XVSM verwenden. Das Ziel dieser Masterarbeit ist nicht, eine vollständige Dokumentation des derzeitigen Status von XVSM zu erarbeiten, sondern beim Leser das Interesse zu wecken, XVSM zu verwenden und darüber hinaus auch ein Basiswissen über seine Features, Möglichkeiten und seine Anwendung zu vermitteln.

Coding standard

In order to have code parts with uniform layout, I have decided to use coding standards which gives guidelines how code must look like to be used in the Tutorial. These relate to common coding guidelines used by the Java programming community:

- Class names are nouns or a sequence of nouns, whereas each noun is starting with an upper case character: SalesPerson, CapiFactory,…
- Interface classes need to start with an I (upper case i), followed by the "normal" class name: ICapi, ISalesPerson,
- methods adhere to the same rules as Class names, although the very first character is lower case: payTicket(), waitForTicket()
- after the method header of a method, there must be a new line before the “{“
- a trailing “}” at the end of the method or at the end of the class must be in an extra line, where no other character is standing.
- in each way, where the “{“ and “}” are enclosing another code part, this code part starts with a new line (thus it is not in the same line as the “{“) and it is indented by 1 tab.
- the width of the tabstop is 2 characters (as we're using a font with fixed size)
- the parameters in methods are separated by a colon, followed by a single blank. It is up to the developer, if the parameters are standing in one line or in multiple lines.
- Comments have to be written above "tricky" or very interesting parts of the code as well as above methods and class names. It is up to the developer if she/he uses block comments or inline comments.
- For simplicity reasons, it is allowed to handle all Exceptions through one “try .. catch(Exception e)” block. Note that we discourage such error-handling in reality

How to maintain the tutorial

This chapter helps to update the Tutorial when the XVSM or MozartSpaces implementation changes.

Below, you find a list of keywords that is used in the Tutorial. I just listed those keywords that are really used in the text. I left away those that are just mentioned (like Exceptions etc), examples (they are using nearly everything) and the Summaries, as they are mentioning nearly all keywords of this chapter:

AbstractAspect: 4
Aspects: 2.1, 4
  authentification: 4
  automatically making persistent: 2.1
  logging: 4
  notification: 4, 5
  pre-/postMethods: 4
AtomicEntry: 2.1, 2.2, 2.3, 2.4
  getValue(): 2.2
Capi: 2.1, 3
  used to access data: 2.5
  addAspects: 4
  commitTransaction: 3
  createContainer(…): 2.1
  createNotification(…): 5
  createTransaction(): 3
  destroy: 2.5.2
    block if no Entry in Container: 2.5.2
  lookupContainer: 2.1c
  read(...): 2.1, 2.4, 2.5.1, 5
    block if no Entry in Container: 2.5.1
  rollbackTransaction(): 3
  shift(…): 2.1a, 2.5.5
  shutdown(boolean clearSpace): 2.1
  take(…): 2.3, 2.5.3, 5
    block if no Entry in Container: 2.5.3
  write(…): 2.1, 2.3, 2.5.4
    block if Container full: 2.1a, 2.5.4
Container: 2.1
ContainerNameOccupiedException: 2.1c
ContainerRef: 2.1, 2.5, 4
  parameter in data access: 2.1, 2.5
Coordination parameters: 2.1b
Entry: 2.2, 4, 5
  EntryTypes: 2.1, 2.2
  getEntryType(): 2.2
FifoCoordinator: 2.1b, 2.3
FifoSelector: 2.5.1
ICapi: 3
IContainer.INFINITE_SIZE: 2.1a
ICoordinator: 2.1b, 2.3
IExplicitCoordinator: 2.1b, 2.3, 2.5.1
IImplicitCoordinator: 2.1b, 2.3
INotificationListener: 5
  notify(…): 5
IPoints: 4
Implicit coordination: 2.1, 2.5.1
KeyCoordinator: 2.1b, 2.3
KeySelector: 2.4
LifoCoordinator: 2.3
Notification: 4, 5
NotificationTarget: 5
RandomCoordinator: 2.1b, 2.3
Selector: 2.2, 2.3, 2.4, 5
  CNT_ALL: 2.4
  contructors: 2.4
  count parameter: 2.4
Timeout: 2.5.1, 2.5.2
Transaction: 2.5, 3, 4
  passed as parameter when accessing data: 2.x, 3
  pessimistic, behaviour etc: 3
Tuple: 2.1, 2.2
  iterator: 2.2
VectorCoordinator: 2.3
VectorSelector: 2.2, 2.4

I refer to other chapters in:

Chapter	referring to
2.2	example at the end of chapter
2.4 KeySelector	example at the end of chapter
2.5.1	chapter about Selectors
2.5.2	adheres to same rules as read

Application scenarios

Version 1.2

Written by Michael Wittmann

Table of Contents

Chapter 1: The Change in the paradigm 9

1.1 From Client-Server … 9

1.2 … to p2p networks … 12

1.3 … to Space-based data storage 13

1.4 „Being more honest“ 16

Chapter 2: Usage Possibilities for XVSM 18

2.1 Adding more computation power 18

2.2 High-priority requests 20

2.3 Execution path 21

2.4 Execution path with load balancing 25

2.5 Execution path with automatic load balancing 28

2.6 Execution path with recovery after failure 34

Conclusion 37

Chapter 1:
The Change in the paradigm

1.1 From Client-Server …

When designing software, the designer must decide what architecture to use, especially if the requirement exists to distribute the data over multiple computers. There are abundant possibilities to design the software in a way that fulfils this requirement. One of the most common techniques is the **Client–Server architecture**. In this architecture, there is one server which holds the information. On the other hand, there is a (possibly big) number of clients that need information that the server has, so they contact the server to retrieve that information (see fig. 1.1).

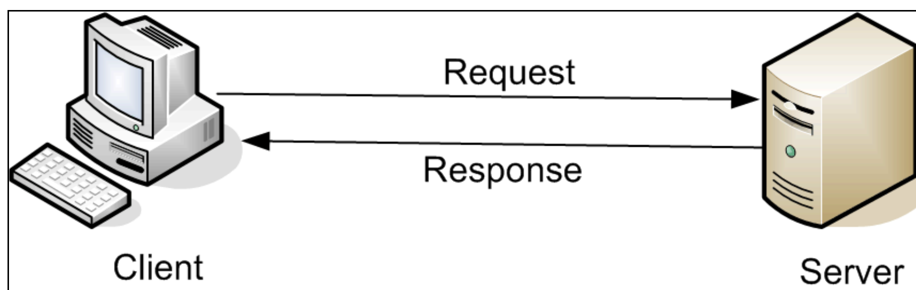


Figure 1.1: Client-Server architecture

This leads to some problems, which are already well-known by most developers, who are using the Client–Server architecture: when the server **crashes**, the information offered by the server is not available any longer. It is even worse as the server typically doesn't just offer data services but it offers more sophisticated services to the clients. So if the server crashes, the complete service is unavailable. Imagine a factory where the server is the central point that sends commands to the production machinery. If the server crashes, all machines would possibly stand still which could lead to heavy losses for the company.

Another situation is that the centralized server does not crash, but is unavailable. For example, it might happen that a server is heavily **overloaded** with requests. The server needs some time to receive a request, parse it, retrieve the information and send the answer back. In many cases, the answer is given back within (milli-) seconds. With a big number of requests, the server might need many seconds or

even minutes until it has processed all requests until being able to answer to the most recently sent request (see fig 1.2).

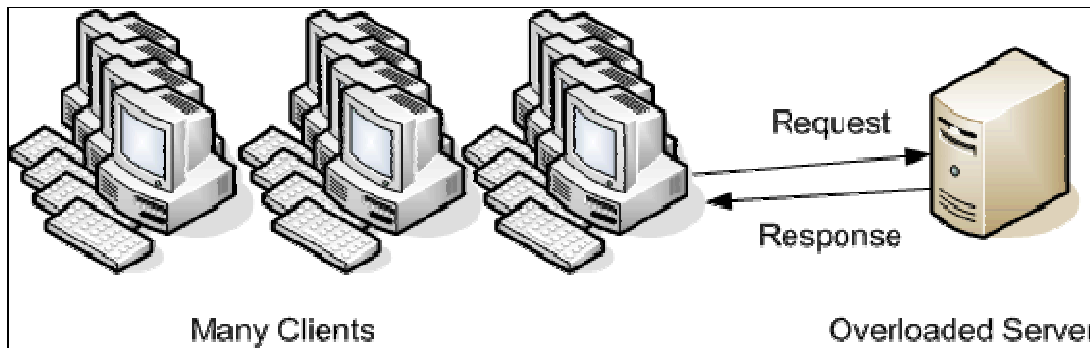


Figure 1.2: An overloaded server

In some networks, especially in the Internet, unreliable communication is used, which means that it is not sure that information is really delivered to the server or back to the client – it could be lost on its way. Thus, there are timeouts used, which give an interval in which the other communication partner must give an answer, else it is assumed that the information is lost. Talking about overloaded servers, which take seconds or minutes to give an answer to a request, this could mean that the server is up and running, still processes an answer, and meanwhile the client **times out**. Normally, the request is sent repeatedly, until the client gives up and decides that the server is down. But resending a request multiple times, that is also processed multiple times, could lead to problems if the request is not idempotent; this means it changes the state of the server on every call. For example, imagine you'd like to order some goods. If the request is sent multiple times to the server, which processes this order, but does not respond within the timeout, the goods may be ordered multiple times. Special treatment for such situations must be implemented.

As a consequence, many companies invest in a **set of servers** to service the clients. There are different approaches how to use a set of servers, like having one server and if it fails, to let the backup server fulfil the requests. Or to equally distribute ("load balance") the requests to all servers to have a good utilization per server and thus a low response time for each request (see fig 1.3)

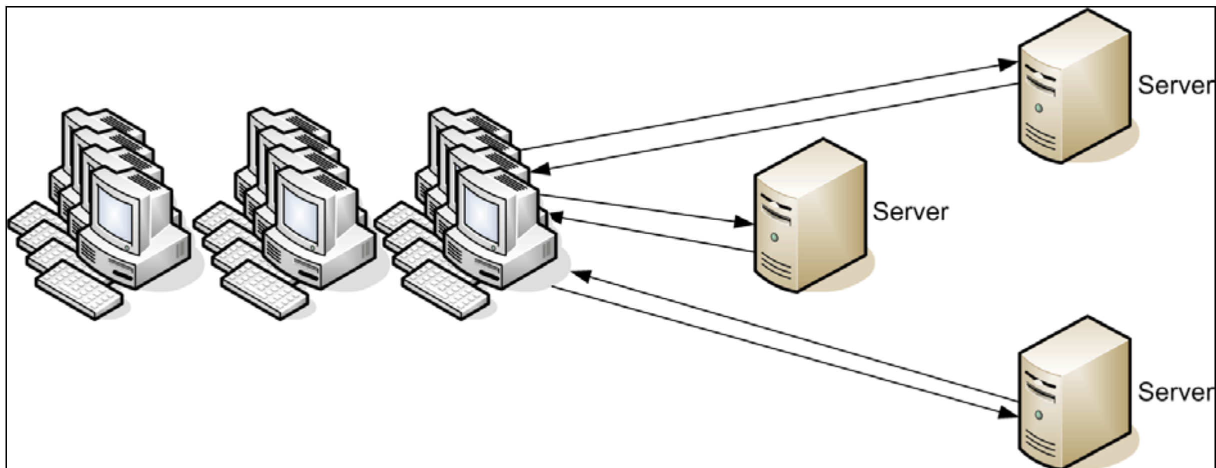


Figure 1.3: Multiple servers answering to requests

In most cases, the servers must react the same way when talking with the clients. The clients are not interested in getting different responses when communicating with different servers. Imagine two servers where one server has one part of the database and the other server has the other one, whereas the requests are handed to a server randomly. In this scenario - depending on to which server the request is routed to - one time the request can be fulfilled and the next time, the information might not be available, as it is standing in the other part of the database. So for this scenario, it is required that the servers must have access to the same information. In many cases, a shared database is used. In this case, the bottleneck is the database itself, as it is now the central point of failure and heavy load. This leads to the same problem concerning bottleneck and failover as described above. In other cases, the servers may **synchronize** their memory directly. Whenever a server has new information, it must communicate this new information to the others. When the server is required to delete data, this server needs to order the other servers to delete this data too (see fig 1.4)

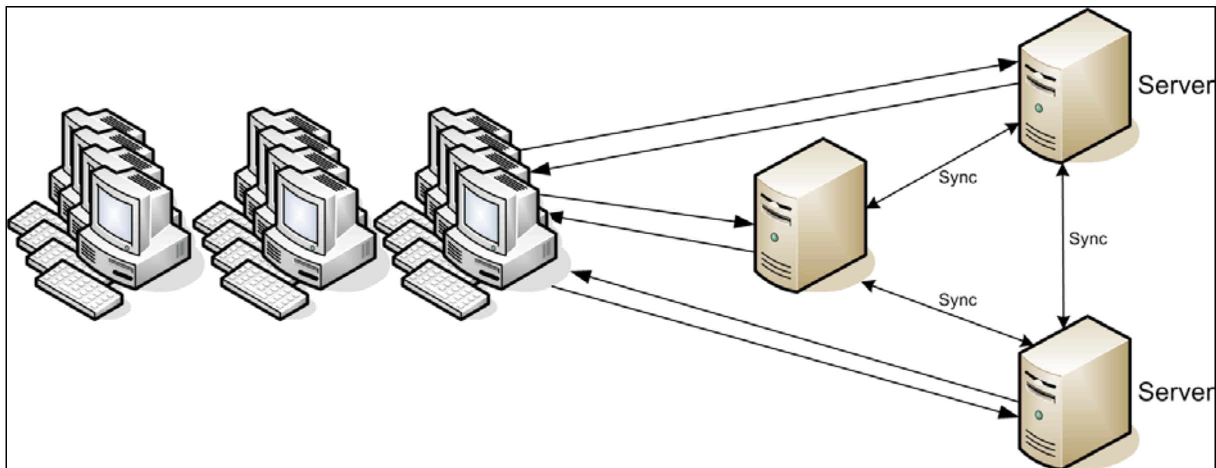


Figure 1.4: Synchronization of a set of servers

However, still the problem remains that servers may fail or become overloaded.

1.2 … to p2p networks …

Recent technologies [1] in offering information for programs are not using dedicated central servers, but instead, each participant in the network is a ”peer”. Therefore such networks are called peer-to-peer (p2p) networks. In such a network, each participant may act as a server and as a client at the same time, in decentralised p2p overlay structures these peers are also called „SERVENTS“ (SERver + cliENT). To be able to answer to requests, the peer must have at least a part of the complete information that might be needed by other peers. A peer can ask a set of peers and try to get the desired information. It depends on the replication and caching strategy of the p2p network [1], how to obtain the data and when to give up, when it seems that the information cannot be obtained within reasonable time. For example, Gnutella [2] asks a group of so-called Ultrapeers that don't necessarily contain the searched data themselves, but store information about successful search requests to peers. Therefore, not all peers need to be asked for information, but it is sufficient to only request Ultrapeers, which in turn ask the so-called leaves, which don't cache the request's responses. The Ultrapeers are iteratively asking other Ultrapeers for the requested information until a certain threshold.

1.3 … to Space-based data storage

Obviously, it would be nice if **all peers** had the **same information**, so either information is held by a peer itself (don't forget that the peer acts as a server as well!) or the information is not available at all – i.e. no other peer needs to be contacted to ask for that information. In most cases, it is impossible to use such an approach, as with the just-mentioned file sharing network, as there would be too much data to be distributed between and stored at the peers.

But this scenario has some interesting advantages as it solves the problem with failing and overloaded servers: if one peer fails, availability is still given. If one or several peers are overloaded, only the user requesting the overloaded peer is affected.

Such functionality can be offered when using space-based data storage. All data is stored within a so-called data space, which the other peers can access. The data space may be spread among a set of peers, which need to keep the data in sync (which is called "replication"). The space only stores the data objects and manages the requests for such data. It also may store the data to a persistent storage device to recover from failure or to enable restart of the system without loss of data.

There are several implementations of such a data space existing, for example Corso [12] JavaSpaces [3], GigaSpaces XAP [4], TSpaces [5], XMLSpaces [7] and XVSM (eXtensible Virtual Shared Memory), although not all of these spaces provide replication or persistency of data:

Name	Replication	Persistency	Transactions
Corso	X	X	X
JavaSpaces		X	X
GigaSpaces XAP	X	X	X
TSpaces		X	X
XMLSpaces	X	X	X
XVSM	P	P	X

Table 1.1: Comparing several data space implementations. X means that this feature is offered, P means this feature is planned

One major advantage of XVSM is what the X in the name is standing for: it is standing for eXtensible, which means that XVSM can be extended by several

functions, like automatic data persistency and recovery, accessing the data by other selectors than the default ones, and so on.

In XVSM, the place where data entities are stored in is called “Container”. One peer can host a set of Containers, this set is called local Space in XVSM:

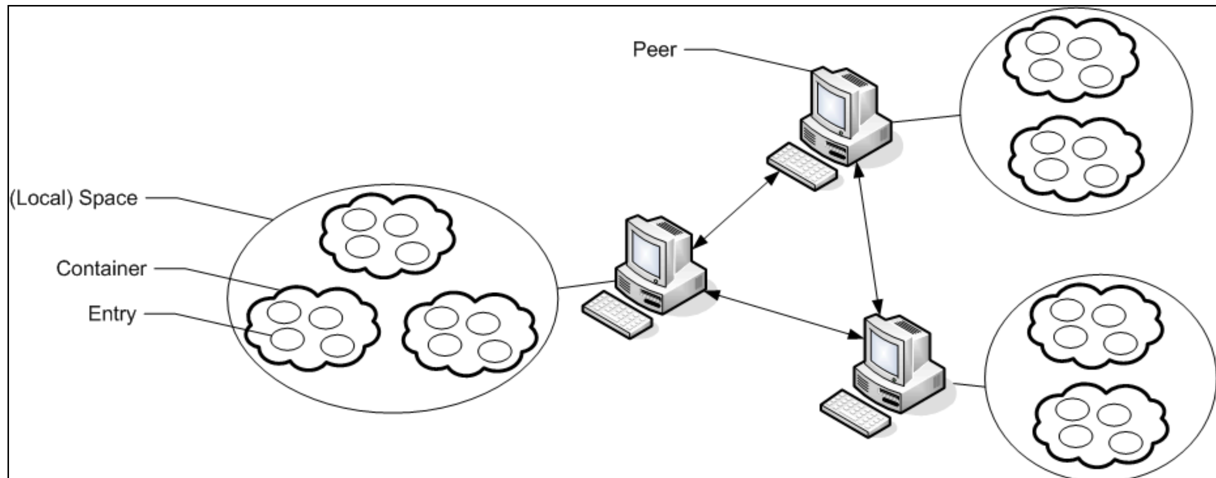


Figure 1.6: p2p Network, each with its own Space and its own Containers and Entries.

Data entities that are stored in a Container are called “Entries”. An Entry can be either an AtomicEntry that is generic and can be instantiated using any Java class or a Tuple, which can contain a (set of) Tuple/s or AtomicEntry/ies. The Entries are written to, or taken, read or destroyed from the Container:

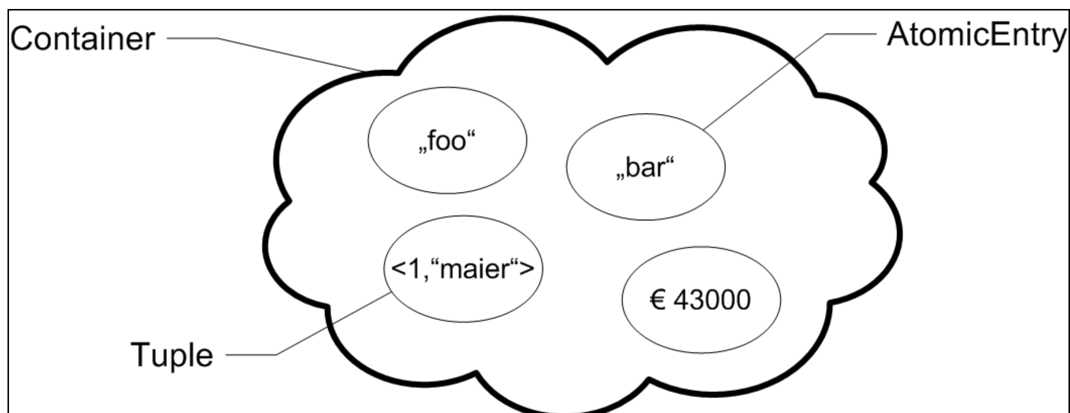


Figure 1.7: Overview Container, Tuple, AtomicEntry

As you can see in fig. 1.7, the Container is used as storage for data entities, which can be considered messages: one peer writes to the Container and wants to tell the other peers something by doing so. Thus, XVSM handles the communication and you no longer need to handle it "manually". The distribution of data is done by XVSM. Note that the data that is stored in the Container can be payload data as well as

command data, even mixed in one Container. Further note that using this approach, you can achieve decoupling in the sense of time and space.

As other peers might need to know about data stored by a peer, they are listening to new information. Such a listener on new information is called in [11] "Producer", which is listening to requests from a peer called "Consumer" [13]. The Consumer writes an Entry to a container, and the Producer takes it. As an answer, the Producer writes an Entry to the container that is taken by the Consumer:

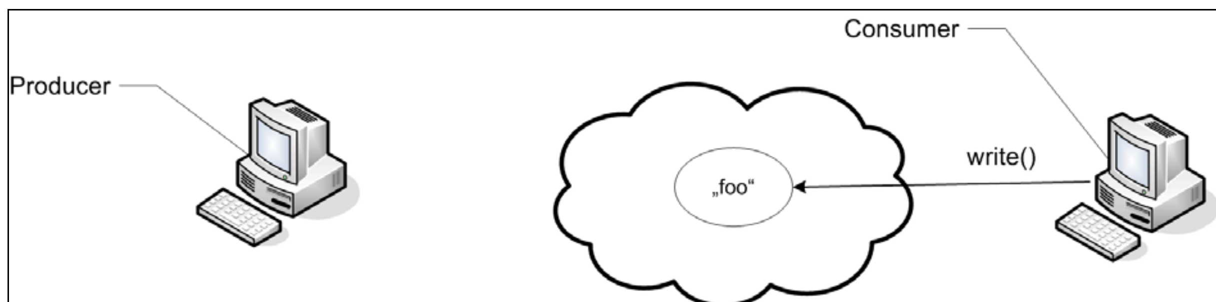


Figure 1.8a: Consumer writes data

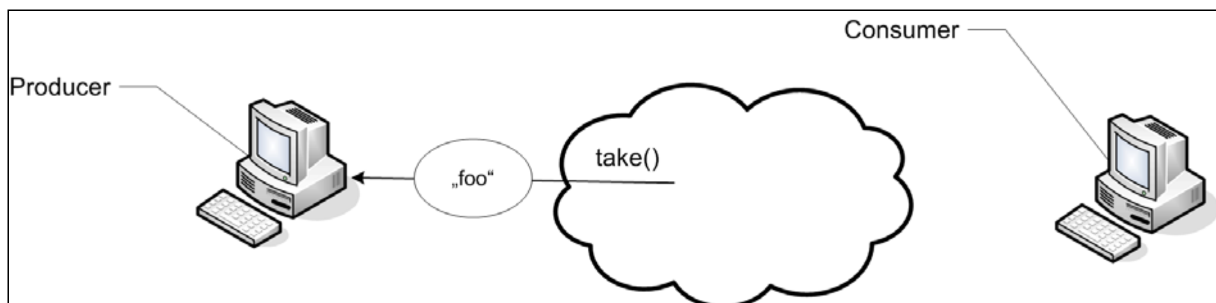


Figure 1.8b: Producer takes data

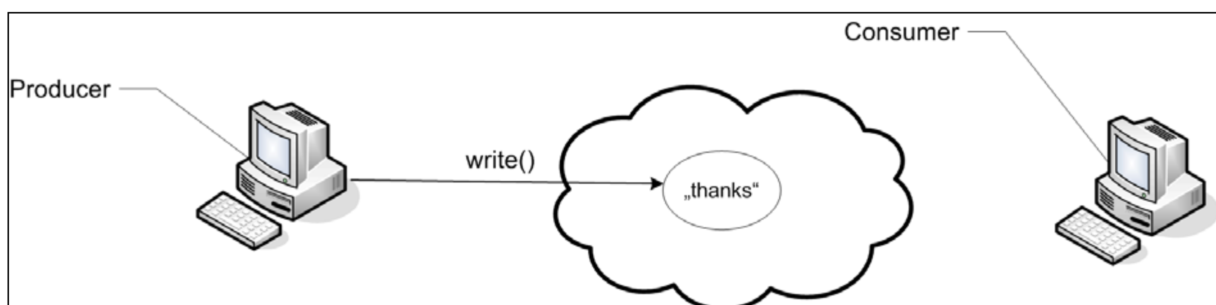


Figure 1.8c: Producer writes data back

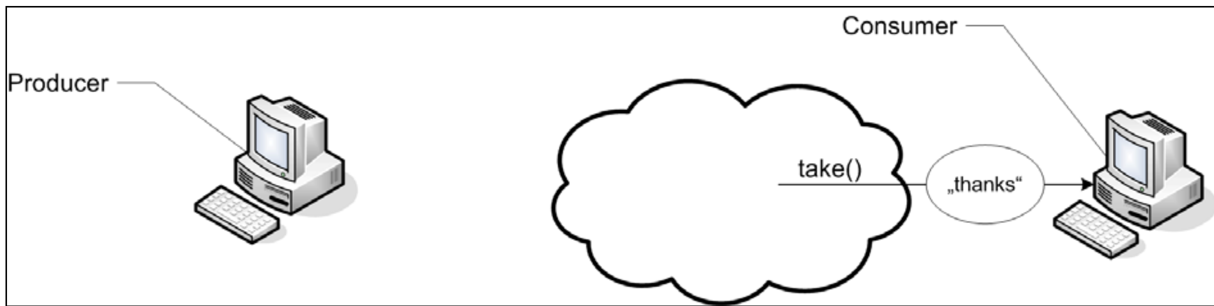


Figure 1.8d: Consumer takes data

XVSM offers the possibility to notify the peer that is working on the Container if a certain Entry is written to the container. Using this functionality, one can implement to push data by using notifications or to regularly pull data from the Container. It is no problem to also mix these approaches within one Container.

1.4 „Being more honest“

Ralf Westphal explains in [11], that techniques like RMI or CORBA hide the important fact that the producer and the consumer are physically distributed – a call to a remote process just looks the same as a call on a local object, i.e. in the same context:

```
String returnCode = producer.foo();
```

Internally, for the treatment of a remote execution, the consumer's method call is given to a communication proxy (e.g. RMI stub), which creates a communication package, and sends it to the producer's communication proxy. This proxy then hands the method call to the producer:

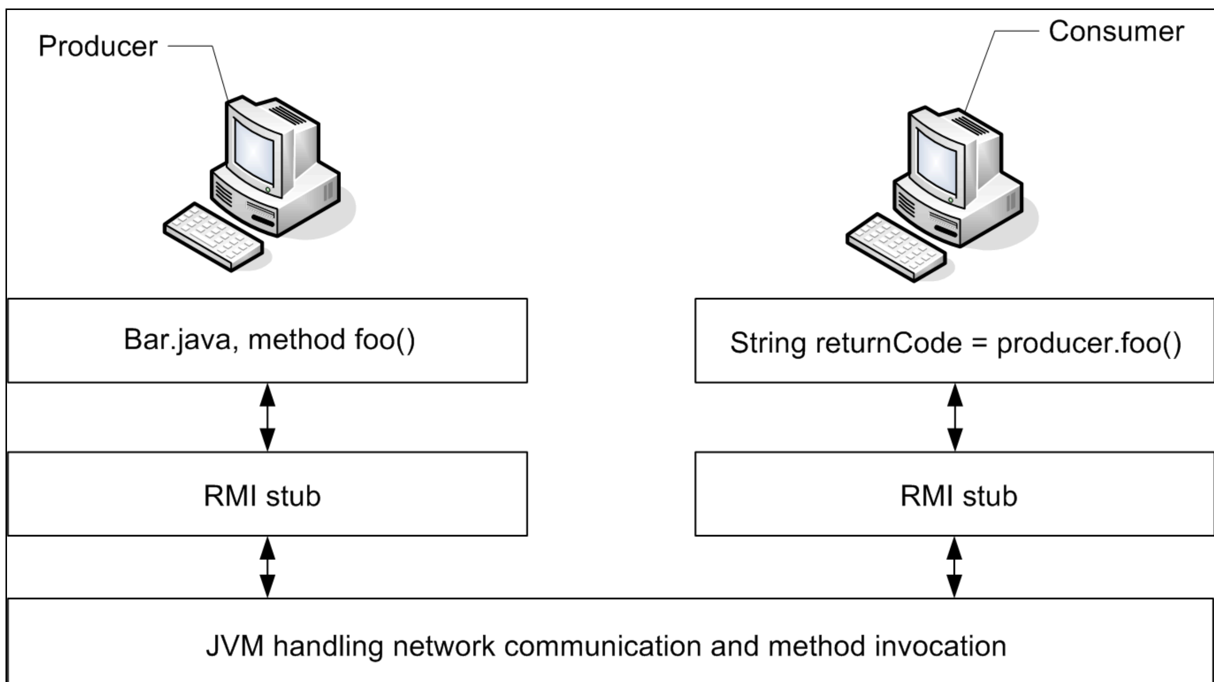


Figure 1.9: Simplified function of RMI

But hiding the details of the communication might lead to wrong expectations at the caller's side and thus according to [11] to "wrong promises". The reasons are:
If you want to perform a local method call, you can be sure of the following assumptions [11]:

- Execution of the request and delivery of the result without any delay
- Handing over complete control of the thread to the producer
- 100% availability of the producer
- 100% secure communication between consumer and producer
- 100% fail-safe communication between consumer and producer
- The execution is performed immediately

Obviously, these assumptions are not necessarily correct for working with remote processes. But as remotely running processes should act autonomously (according to [11]), there should be another paradigm used, and that's where XVSM comes into play: XVSM can be used as a message exchange platform, which holds the requests from the consumers and the answers of the producers. The consumer cannot "command" the producer directly, as the consumer only writes a request to the space and the producer takes the request to fulfil it. The producer autonomously decides to handle a request and it also decides the order of handling requests; this style of interaction is not *imperative* anymore, but *cooperative*:

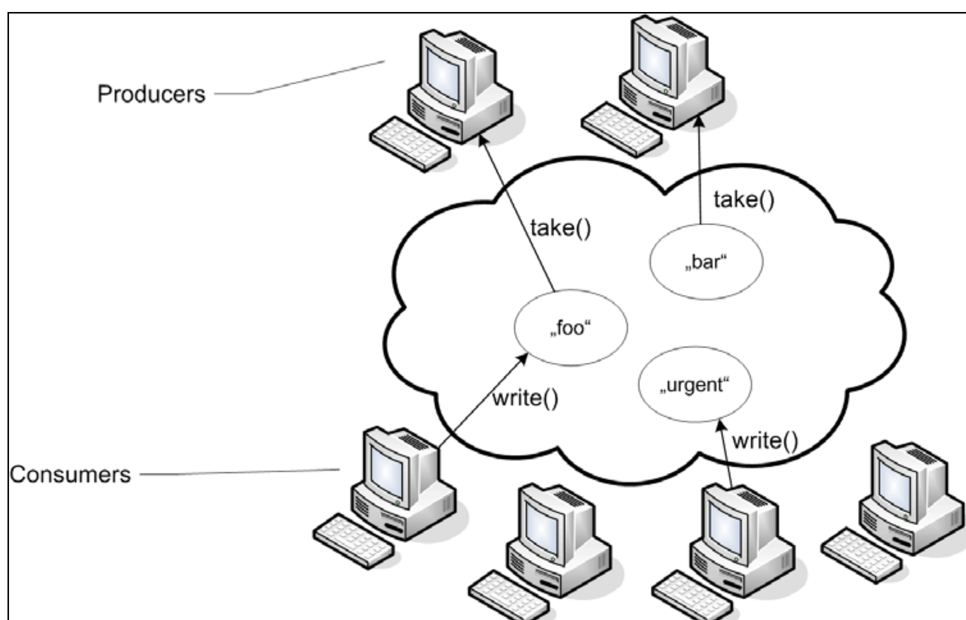


Figure 1.10: Multiple consumers, multiple producers

As XVSM, like other space-based approaches, offers the possibility to let multiple peers communicate using the space, there might be multiple producers handling the requests, and they can cooperate to handle them. On the other hand, the detail that a request might be lost, as no producer is available, is not hidden from the Consumer – it has to provide some possibility to be notified as soon as the request is handled. This might take forever if no producer is available for this request, but it is on behalf of the consumer to take special treatment in such a case, which is more appropriate for operating remotely and is not hidden from the programmer.

Chapter 2: Usage Possibilities for XVSM

2.1 Adding more computation power

In chapter 1.4, the possibility to use XVSM to implement an improved version of the Client-Server architecture towards a “more transparent” way of request processing was shown: the Client is aware of the possibility that no server is available and thus it may take action that its request will be fulfilled within a given time. In fig. 1.10, of course there is still the possibility that Producers fail or that they are overloaded. But as Customers are aware of this behaviour, they can either try again or contact an authority to ask it to add more Producers:

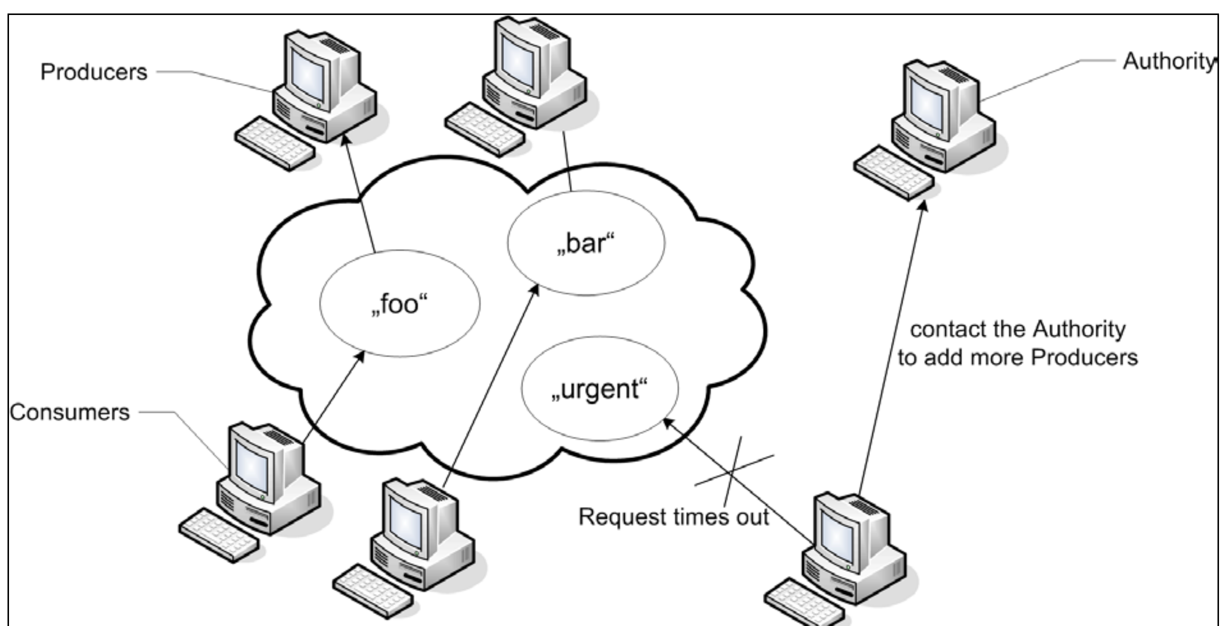


Figure 2.1.1: Request times out, Consumer contacts Authority

The contact to the Authority can be established according to certain policies like e.g. after a certain number of retries. The Authority decides if it is really necessary to add new Producers; e.g. if all Producers are really overloaded. In the case that the Producers are not fully utilized, but the Consumers complain that the Producers are not answering, it might be that the Consumer(s) are not requesting in a way that the Producers are able to react upon. For comparison: Imagine a room with two sales persons and several clients. The sales persons answer questions about product prices. However, assume that there is one client that asks when the bus will arrive. The sales persons won't answer to this question and after a few retries, this client will give up. An Authority now may decide that it's not a problem of the salespersons (as they are not overloaded), but that the client is not requesting correctly only by comparing the utilization of the Producers (which indicates that they are not overloaded) and the fact that requests from Consumers are arriving that there are not enough Producers. In this case, the Authority must provide some means to contact a human administrator to inform about the fact that a Consumer's request cannot be fulfilled. This is in contrast to an imperative approach: Using an imperative approach (e.g. Client-Server architecture), you receive an answer either way, which can be an error message if the request was formatted in a bad way. In some cases, also the cooperative approach (as just described) can handle cases where erroneous requests are detected, but only with limitations. If a request is too incorrect, it may not be processed at all.
In case that the Authority decides that a request is invalid, it can remove this request from the "normal" container and move it to an extra "error container", as proposed in [15]. The human administrator may be notified about such a new Entry or he/she may check this containers content regularly to decide what to do with such Entries.

It depends on the implementation of this Authority, when it decides to add new Producers, and how it is done – but let's say there is a pool of available peers ("Resource pool") and one peer is added to this space just by telling it that it needs to handle (a subset) of requests and giving it the reference to the space container where the requests are stored. The Resource pool may be the internal memory of the Authority or another container with contact information to available peers. When

writing this information to the memory or to the container, the peer is considered being available for a new execution task.

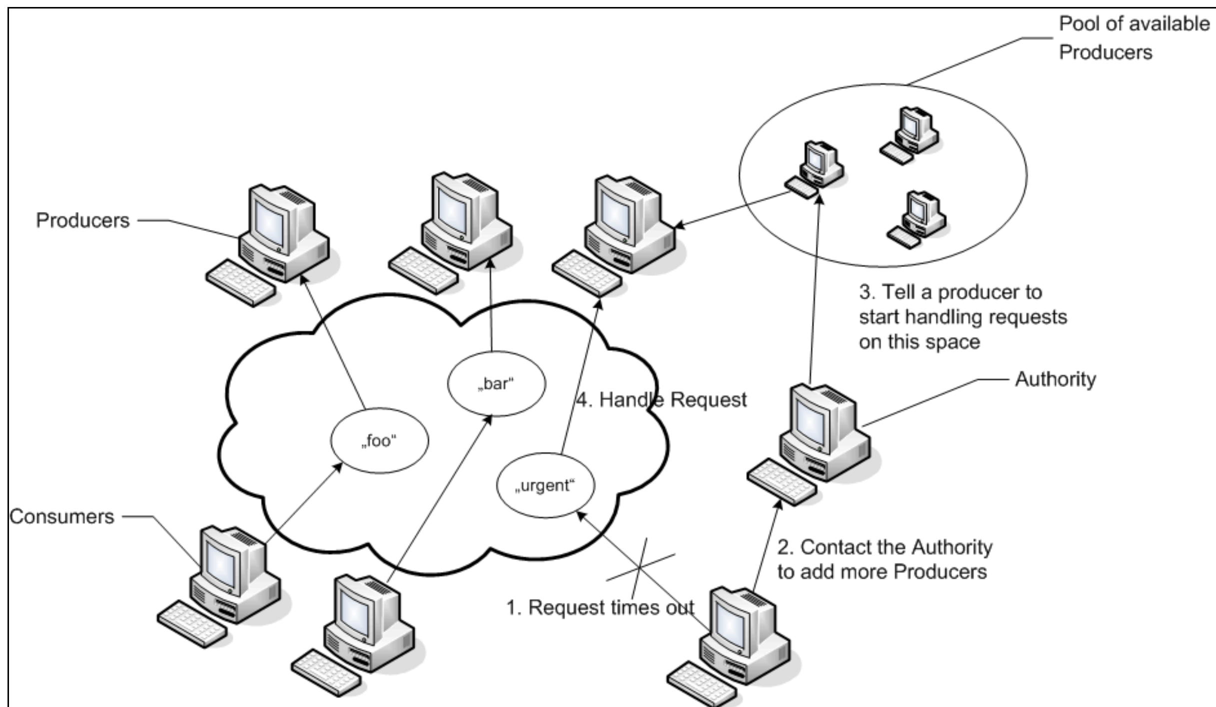


Figure 2.1.2: Adding a new producer

A very interesting property of this scenario is shown in fig 2.1.2: A Producer actually can be a thread in a process that is running as a Producer, a single workstation, a mainframe or a cluster that is running processes that are acting as Producer – it just doesn't matter. It is always only a “worker” that is checking if a certain request is standing in the Container. Of course, if there is no Authority or if there are no more producers available, also XVSM can't help. But keeping in mind that XVSM allows to detect whether a request is not handled, the customer can – in contrast to other paradigms – handle such a situation. For example, when an Entry is currently handled, it is taken from the Container and written to another. If after a certain time, the Entry still is in the Container (it is still not processed), the Client that stored the Entry can take action, like error handling, user notification etc.

2.2 High-priority requests

An Entry may hold data beyond or instead of a request itself (called payload data): so you can use this data to define a priority. It is also possible that the Entry itself is of a type (class) that represents a request with high priority, for example a type that

extends the functionality of the Entry by giving it a "high-priority label". With such a request that is declared as being high-priority, you may implement a system where a dedicated peer listens only on requests with high priority:

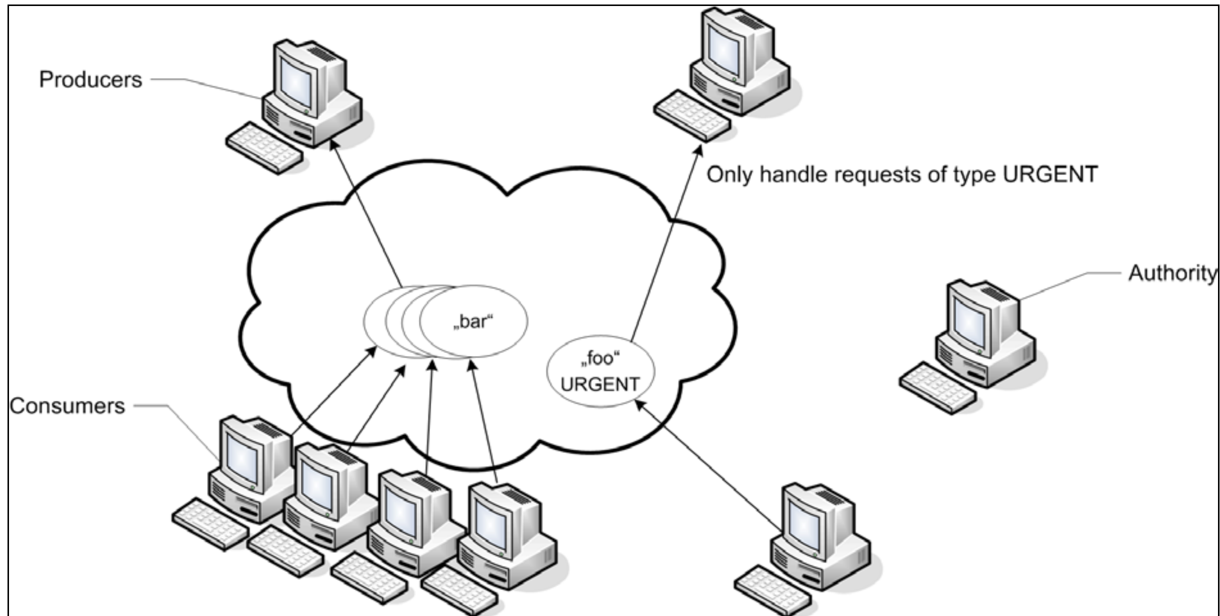


Figure 2.2: One Producer only handles urgent requests

There is also the possibility to implement the high-priority request handling not with a dedicated producer, but in a way that each peer (before taking an Entry from the container) checks if there is a request with high priority. If there is one, the request is executed, if not, a "normal" request is taken.

2.3 Execution path

Building on the examples from Chapter 2.1, we can implement an execution path. An execution path is a sequence of operations, where the output of one operation is the input of the subsequent operation. It has one or many defined Start state(s) and one or many defined End state(s).

We could define an execution path as the following state machine:

- There are Pools that contain intermediate results of an overall operation. The Pools are represented by a state in the diagram.

- The edges are operations that are performed by one or more executing entities on an Entry in the Pool. Such an executing entity is a peer in the space that offers its processing power to perform the operation.
- An operation moves an Entry, which contains the intermediate result of the operation, from one state to the next.
- There are Pools at the start and at the end of the path that contain the initial values resp. the final results.
- The sum of all passed edges between a start and an end Pool are considered as being a complete computation (e.g. the computation of PI, or fulfilling a Sales request etc.).

An example for an execution path could be:

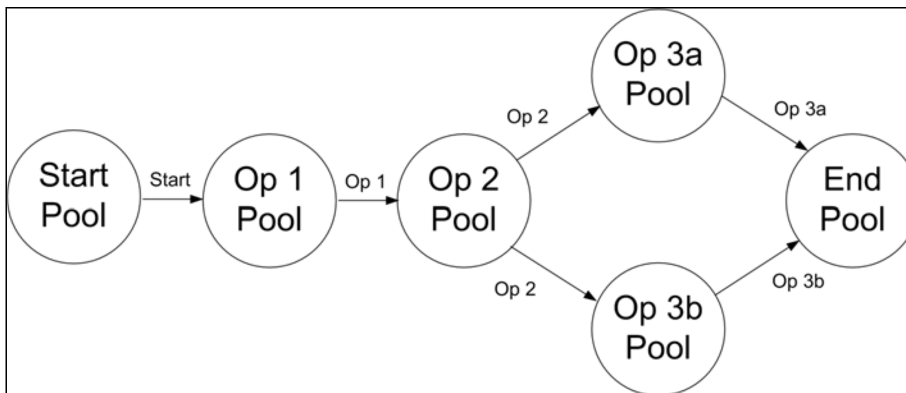


Figure 2.3.1: Execution path

In fig. 2.3.1, the Pools are drawn as circles (states). In the Pools, the above mentioned Entries with the intermediate result are stored. The linking arrows between the Pools are standing for the operation that is needed to bring an Entry from one Pool to the next.

On each operation, some entities are included that perform calculations (processors, clusters, …). After an operation is finished, the result is passed to the next Operation Pool. There also may be – depending on the result of an operation – a branch in the Execution path. Furthermore, there may be multiple start and/or end states:

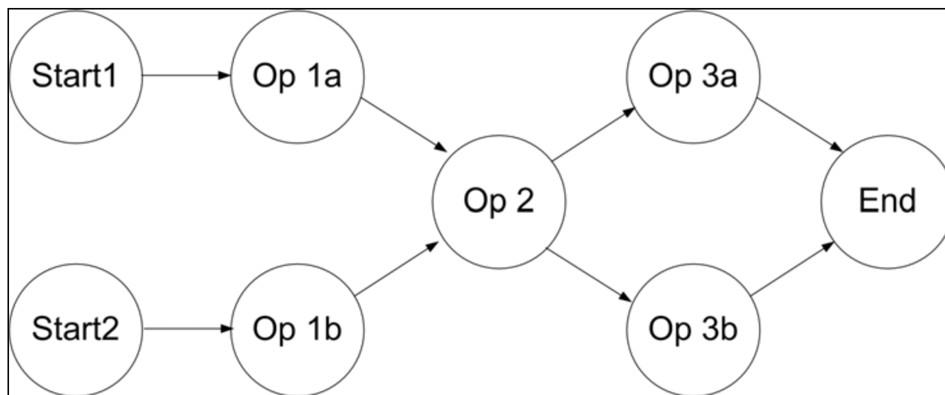


Figure 2.3.2: Execution path with multiple Start states and branches in the calculation from Start to the End

The possibilities to combine the states (operations) are vast.
If you have a calculation to perform, you put your initial value to (one of) the Start states. A calculation entity takes this value and performs an operation on it. The result is passed to the next Operation Pool. You can imagine this taking and passing of the value like a token that moves from one state to the other in the above state diagram, which is also known as the SEDA approach [6]. One of the advantages of the SEDA is that multiple peers can work on one or different states in parallel and independent from each other. Thus, in heavy load conditions, it can perform better compared to a thread-based approach, where each thread executes the complete execution path without any intermediary states [6].
Let's say that the token that represents an intermediary result is an Entry in XVSM and each of these pools which hold the intermediary results is a container. Peers are listening to new Entries in a specific container. A peer takes an Entry and performs an operation on it. When it has finished, the peer writes the result to the next container (state). There, it is taken by a (possibly other) peer and so on, until all operations are done and the result value is written to the End state.
Each container holds a number of Entries, which are the intermediary results. One can now check these numbers per container to see the current progress of the calculation and also see the number of peers that are working on this container. A tool that visualizes the number of working peers and number of Entries per state is implemented in [14].

Obviously, each operation needs at least one peer to perform this task, else, there would be an increasing number of Entries in the container without a calculating peer

and there won't be any result. But of course there could be multiple peers operating on one container:

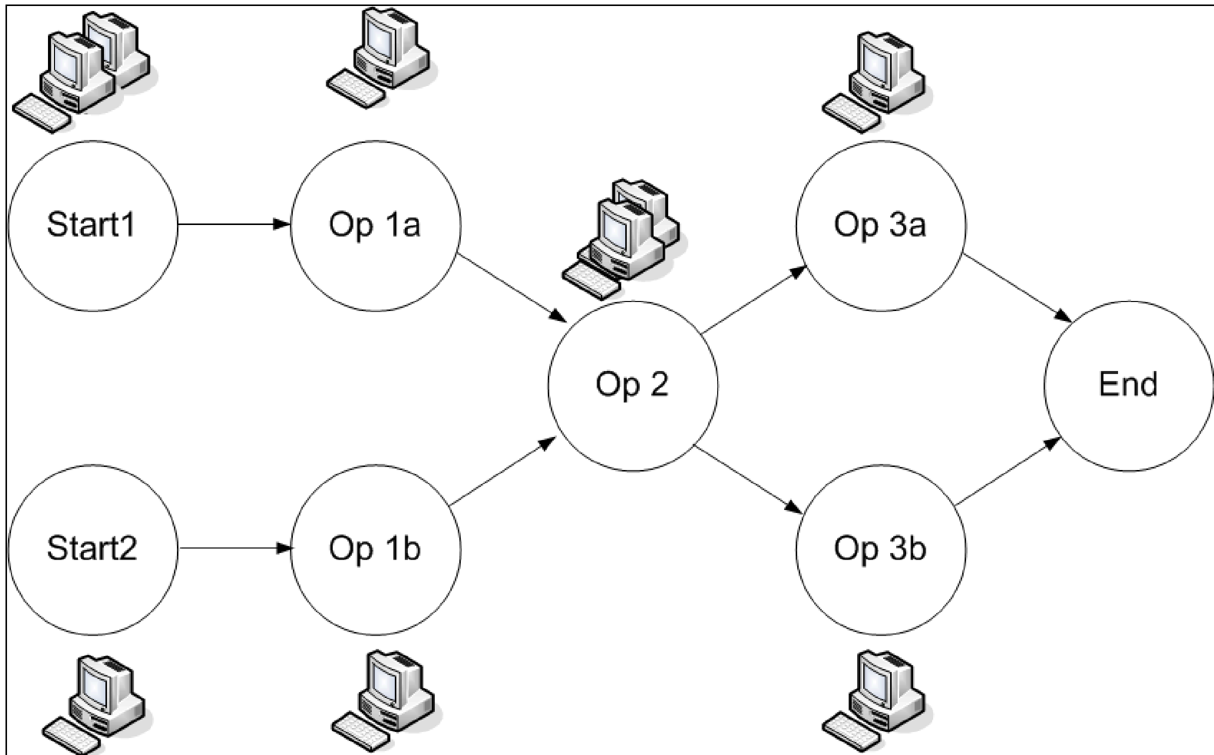


Figure 2.3.3: Execution path with multiple peers working on one container

The Entry already contains the algorithm as additional payload, for example as a script. Thus, the peer takes the Entry and performs the operation that is common for this one container. The information which operation is specific for which container may stand in the contained algorithm. In this simplified example, it is only assumed that there is some possibility to store the algorithm within the Entry.

Keep in mind that an Entry could be an AtomicEntry, which is generic. In this AtomicEntry, you have some object stored, that holds the current value and describes the operations as well as what to do if an operation is finished, i.e. which operation should be done next. Beside the complete algorithm it furthermore has an internal state that tells which operation should be currently performed. Peers participating in this Execution path only need to retrieve the algorithm that is specific for this container and check what to do with the Entry when the operation is finished, which must also be included in the Entry's algorithm.

Please take a look at Figure 2.3.2. A peer takes such an Entry from the Start 1 container and executes the algorithm contained in the Entry. The algorithm must describe which operation to execute first. This algorithm is executed by the peer. When the algorithm that is specific to the current state is finished, the internal state of the Entry is set to let the second method perform next. But this method is not performed by the peer immediately. Instead, the peer checks by usage of the return parameter of the current algorithm that the Entry must be written to the subsequent container. Thus the peer does what the Entry tells it and writes the updated Entry to the Op 1a container. There, the Entry is taken by a peer (either the same or another one), the algorithm which is specific to container Op 1a is called and so on.

Each container now holds a number of semi-products (Entries where the calculation is not finished yet). As the peer doesn't know the algorithm in advance and doesn't know what kind of Entries are standing in the containers, the peer doesn't need to be prepared specially. It just needs to execute the algorithm passed with the Entry. All the operation instructions are stored in the Entry itself, the peer just performs the operation. Using this approach, you can use a peer for any operation state, as the peer doesn't care what container or Entry it is currently working on, it only performs the actions told by the Entry. Security issues are not taken into consideration in this simplified example, but of course may be an issue in a real-life project.

2.4 Execution path with load balancing

Let's assume we assign exactly one peer to each container. Then, as already explained, the Entry is passed from one peer to the next. Furthermore, we assume that each operation takes equally long and that there is the same number of initial Entries written to Start 1a and Start 1b. But in Op 2, there is only 1 peer listening for new Entries. Thus, if the Entries are coming in quite fast, this peer will be overloaded and it cannot perform the operations in time. The preceding peers are writing their results to the Op 2 container and thus, this container gets more and more Entries.

Please keep this scenario in mind as it will be referred to again in Chapter 2.5.

We already know from Chapter 2.1 that with the usage of a Resource pool and an Authority, a new peer can be added to "support" other peers with their calculations. The Authority regularly checks the number of Entries in each container. If there is a high number of Entries in a container that exceeds a configurable threshold and if there are resources available in the Resource pool, resources are added in the form

of peers to the container. Note that such a processing resource could be of different types (thread, processor, mainframe etc.). The processing resource (a.k.a. peer) is told that it now is listening to Entries in a specific container. As soon as an Entry is written to the container, the Entry must be taken and its algorithm executed. In the scenario just described, where in Op 2 an increasing number of Entries will occur, it is appropriate to add a peer to this container to perform the operation. It is up to the Authority to decide, (a) which peer should be added to the container, and (b) at what load per container the peer should be added. Of course, if there is a low number of Entries in a container, the Authority could decide to take peers away from the container and give them back to the Resource pool ("shrink" vs. "expand"):

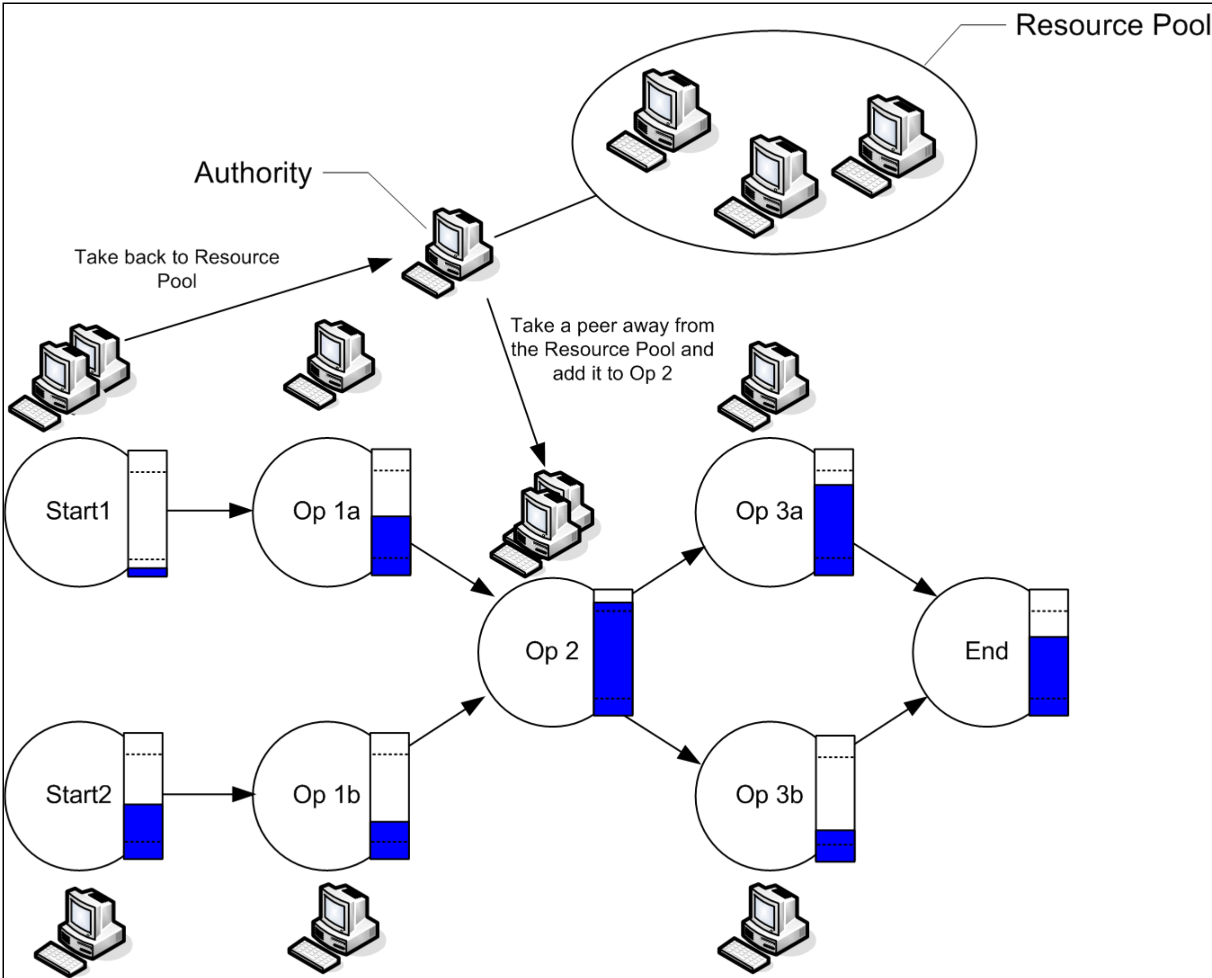


Figure 2.4.1: Execution path with an Authority to perform load balancing

The symbols used in fig. 2.4.1 and subsequent figures represent the following:

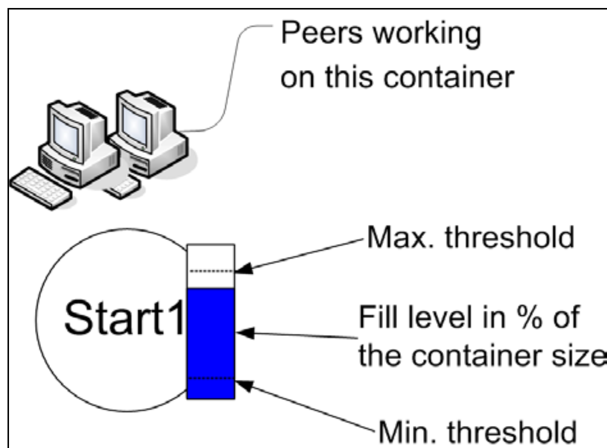


Figure 2.5.1b Symbols used in the previous diagram

Again, please note that the resources, the container and the Authority are independent of the operations that are done and of the Calculation itself, as all the information about the operations and calculation is contained in each Entry. This scenario can of course also be accomplished by many other programming frameworks. But XVSM has an advantage in comparison to the some other ones: an Aspect can be implemented that performs the necessary calls to listen to a container, executes the Entries' algorithm and writes the Entry to the container of which the Entry knows that it needs to be written to next. You register this Aspect at an XVSM peer that wants to join the Execution path and that's all. Other middleware systems that offer similar functionality to implicitly react on performed actions is for example Lime [19], which offers the possibility to register reactions on actions that are performed on the space.

The Authority can be implemented in a way that it is called by a peer, if the peer notices in some way that the upper/lower threshold of container entries is exceeded that shall trigger addition/removal of Entries in a container; or the Authority can check the various containers time by time by itself and decide if peers should be added or removed from a container.

We assumed that each operation takes equally long and that into Start 1 and 2 the same number of initial Entries is written. But it is quite obvious that this may not be realistic. operations can have different duration and the initial values are not written with same frequency. But this doesn't matter: Let's say that there are twice as much Entries written to Start 1 than to Start 2. On both containers, only one peer is

listening. After some time, there might be so many Entries in Start 1, that the Authority adds one more peer to Start 1. Then, there will soon be too many Entries in Op 1a, so the Authority adds here one more peer too, and so on. After some time, let's assume that no more Entries are written to Start 1. There are currently two peers working on this container, so eventually there will be no Entries in this container any more. These two peers notify the Authority that the container is empty, so the Authority withdraws one (or even both) from the container and returns them in the Resource pool as described in Chapter 2.1. When the peers are in the Resource pool, they in turn can be added as a worker to another container again.

If you want to implement the execution path, it is not necessary to have a single container for each intermediate result, but you may have only one container for all Entries, no matter what internal state they have. Peers accessing this container are filtering the Entries by their state and taking those Entries that they are listening on. For simplicity reasons, the style with one container per intermediate state, as already described above, is used.

2.5 Execution path with automatic load balancing

One thing that is still needed in the previous scenario is the Authority that adds or removes the peers from the containers. But it is possible that the peers are managing this by themselves.

Take the same scenario as from Chapter 2.4, but without the Authority, and assuming that the containers are limited in size. There is only one well-known peer running that holds a container that represents the Resource pool. If a peer decides to request more computing power for the container it is currently working on, it writes a request to the Resource pool (which is a container, too). If a peer decides that there are enough working peers on its container, it could decide to "leave" this container. In this case, it gets one Entry from the Resource pool, which is a Request for Help. From now on, the peer listens to Entries in the container where help was requested:

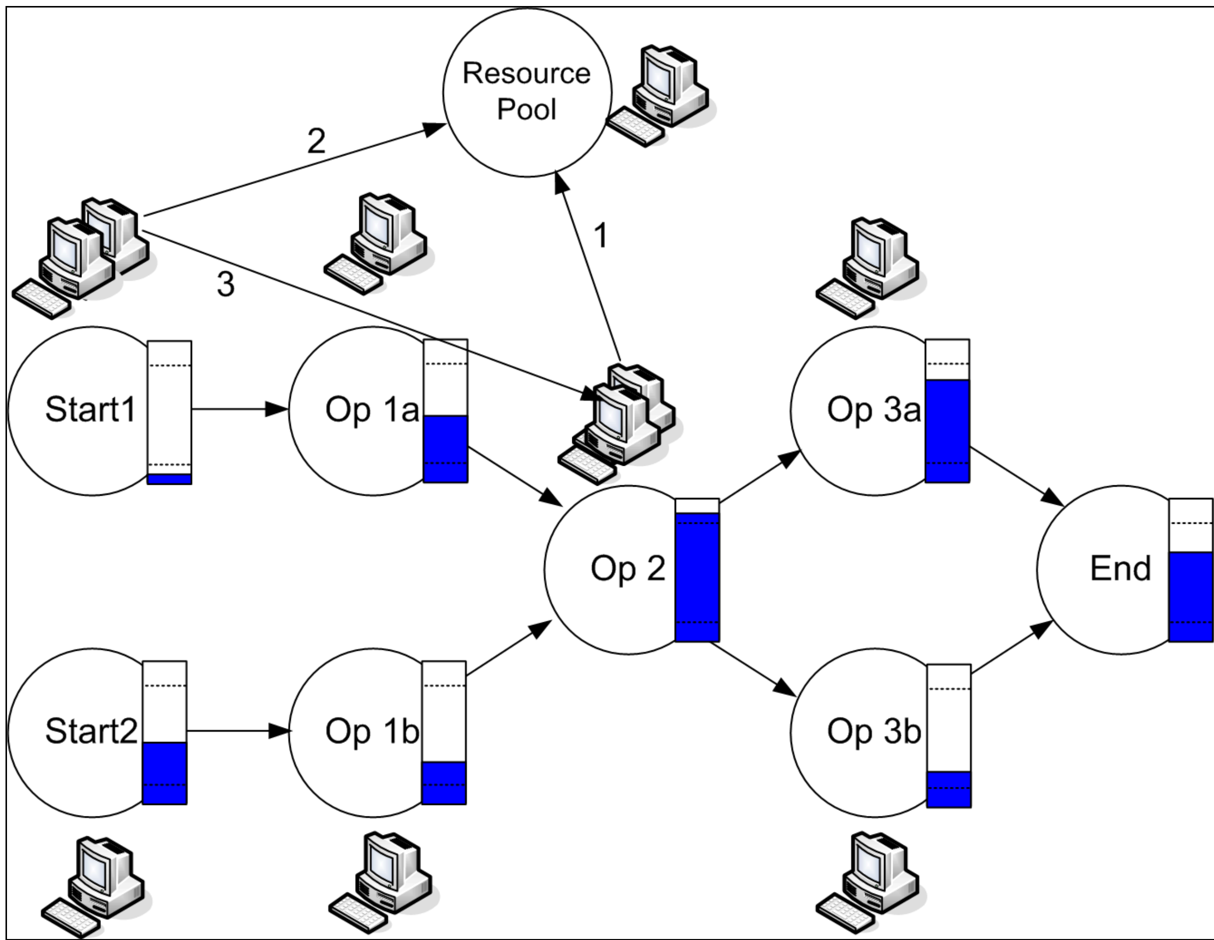


Figure 2.5.1: Execution path with different fill levels of the containers

1. The maximum threshold is reached, so the peer that is working on Op 2 writes to the Resource Pool an Entry that holds a request for help. The request for help must be unique for a container (usage of keys), else many peers per container could write the same request many times to the Resource Pool.
2. A peer decides that the minimum threshold is reached and thus, it doesn't need to work on this container any more. So it reads the requests in the Resource Pool. If there isn't any request, the peer either blocks (as it is done if there is no Entry in a container) until a request is written or it returns to its old container, waits for Entries and regularly checks if there are new requests in the Resource Pool. But as in our example scenario there is a request in the Resource Pool, the request that Op 2 needs help, is taken.
3. Thus, the peer joins Op 2, and starts taking Entries from there and executing them.

The approach described so far has two – yet – unsolved problems:

- The peer that possesses the Resource Pool crashes
- "Elopement from an empty container": All peers on a container notice more or less at the same time that the minimum threshold is reached and so, each of them retrieves a request from the Resource Pool, so no peer is any longer working on this container.

The first problem can be circumvented by detecting the crash of the Resource Pool peer. This is not hard when using a separate container for the Resource Pool, as a peer that wants to contact the Resource Pool will notice that it is unavailable by the standard network exceptions. In this case, the peer that detects the failure of the Resource Pool will take over its role by creating the Resource Pool container, which must be well-known by all participating peers. If you decide not to use separate containers for each Operation Pool but instead to write all Entries to one container, you also may decide to write the Request for Help Entries to this container, too. In this case, there is no network exception because the container representing the Resource Pool is part of another container:

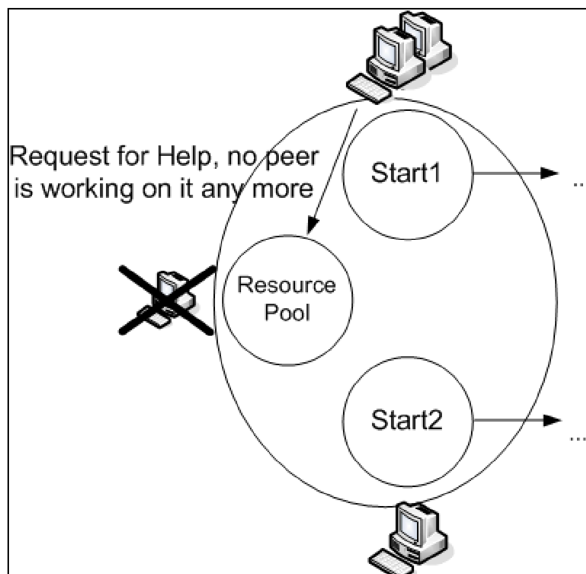


Figure 2.5.2 Peer that handles Requests for Help has crashed, the container itself is up and running

In this case, there is no dedicated peer that only works to handle the Requests for Help messages which could crash. The peers themselves are writing and taking the Requests for Help, thus the peer holding the Resource Pool only needs to offer a container where the Request for Help messages can be stored in. To conclude, if the

container containing the Request for Help messages crashes, then there is a problem, but this can be circumvented as it is easy to detect, as just described.

The second problem, where peers may elope from a container that has reached a minimum threshold can be fixed in the following way: Instead of deciding that the peer itself leaves the container, it stores in this container a message to tell the others that they should leave. Every peer that is noticing the minimum threshold is reached needs to check if such a message exists, except the one that just wrote this message to avoid to let the peer that wrote the message also read its own message. There must only be exactly one or none of such an Entry (by using keys for example), else it might be that all peers at once write such Entries and then none will leave. So, before writing an Entry to leave the container, a peer needs to check if such an Entry already exists and if so, it needs to take this Entry and leave the container.

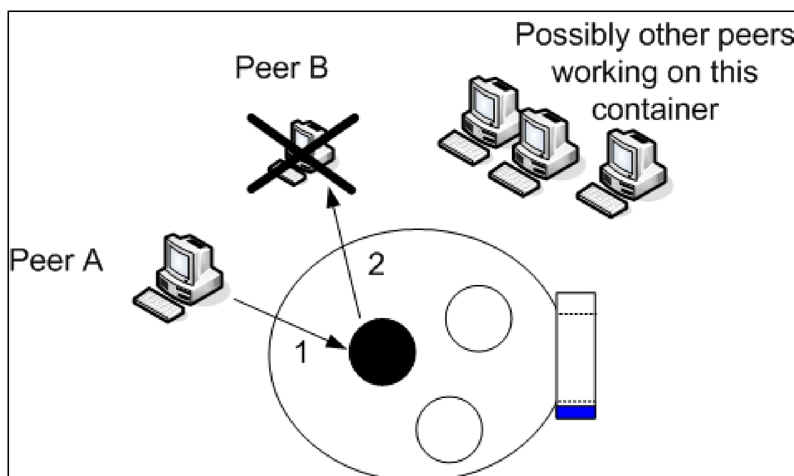


Figure 2.5.3 A peer notices that the minimum threshold is reached and writes the "Leave container" message

The black circle is the unique message (Entry) to "Leave container". The white circles are the Entries to perform the "normal" operation.

1. Peer A notices that the minimum threshold is reached. It checks if the "Leave container" message exists, but as it doesn't yet exist, it writes this message. It no longer listens on a message to leave; else it might be that it receives its own message.
2. Peer B also notices that the minimum threshold is reached. It checks if the "Leave container" message exists. As it arrives (created by Peer A), Peer B

destroys the message and leaves the container (it is returned to the Resource Pool or retrieves a Request for Help, dependent on the approach you chose)

3. The other peers are not included in this scenario, but they might perform similar actions as described in point 1 and 2.

Still, the following scenario could occur:

1. Peer A and B notice that the minimum threshold is reached.
2. Peer A obtains a lock on the container and writes the "Leave container" message.
3. Peer B also wants to obtain this lock, but it is blocked until Peer A commits and unlocks
4. Peer A is no longer listening on "Leave container" messages, else it may be that it receives its own message, which may lead to Elopement again.
5. Peer B notices that there already is a "Leave container" message, takes it and leaves the container
6. Peer C notices that the minimum threshold is reached, sees that there is no "Leave container" message and thus writes one, and furthermore is no longer listening to these "Leave container" messages.

In this scenario, Peer A and Peer C are still working on the container but no longer listening on the "Leave container" messages. To prevent this behaviour, the Peer that no longer listens to a container checks in random intervals if there is such a message. If so, it destroys the Entry and listens again on it. If no such Entry is available, it just listens on its occurrence again to give another peer the possibility to tell to leave this container. By regularly checking the minimum threshold and performing the described tasks, the number of peers working on this container will not immediately decrease to 1, because of race conditions on the regular checks and on taking the entry, but in this case this is an advantage, as then a set of peer remains working on this container until an optimal balance is reached.

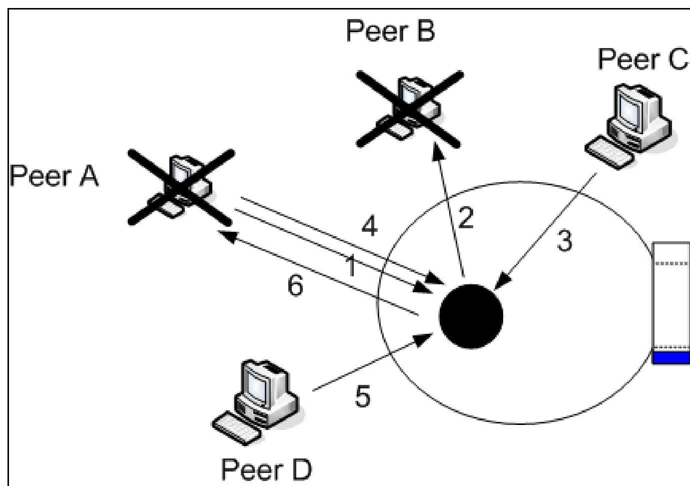


Figure 2.5.4 Peer A first writes the "Leave container" message but later listens to it again

1. Peer A writes the "Leave container" message, doesn't listen on its appearance any longer.
2. Peer B notices the minimum threshold, checks for the "Leave container" message, destroys it and leaves the container.
3. Peer C also notices the minimum threshold, checks for the "Leave container" message (which already was destroyed by Peer B) and thus it writes a new "Leave container" message. It also is no longer listening for this kind of message.
4. Peer A and C now check after a random interval if a "Leave container" message exists. Peer A checks first, it notices that there already exists such a message, so it destroys it and listens on these messages again.
5. Peer D notices that the minimum threshold is reached. It checks for a "Leave container" message which currently doesn't exist, so it writes it to the container.
6. Peer A checks for the "Leave container" message again, which now exists and thus it destroys this message and leaves the container.

As seen in the scenarios just described, a peer regularly checks in random intervals if the minimum threshold is reached. If so, the peer checks if there already exists a "Leave container" message. If there is already such a message, the peer destroys it and leaves the container. If there is no such message existing yet, the peer creates it. The peer sets an internal state that it already has written this message. After a random time, again, this peer checks if the minimum threshold has been reached. As

still, this is true, but the peer already has written the "Leave container" message (internal state!), the peer only checks if the "Leave container" message exists. If it does exist, the peer just destroys it. No matter if the message exists or not, the peer has no longer the internal "I've written a 'Leave container' message" state and thus is agreeing again to leave the container if such a message appears.

You also can implement the load balancing in a way, where the Requests for Help have a priority, depending at which level the maximum threshold was reached. Doing so, the container (state) that needs help most urgently is served first when assigning peers to containers.

2.6 Execution path with recovery after failure

Until now, we always assumed that each peer that is working on a container always works “perfectly” in means by its reliability and availability. But as we all know, programs and computers are not always working “perfectly”. It could be that a peer is shut down during its operation or that someone writes malicious code in the Entry's contained algorithm that lets a peer crash. Also, network errors can occur that let a peer temporarily be unavailable.
Some problems can be solved using the scenarios that are described previously, others can't. In the following you get detailed information about what problems can be solved and how they can be solved.

The containers that represent the Operation Pools are limited in size. Thus, if a peer has finished its operation and wants to write the result to the target container, which is already full, the operation blocks. This means that also this preceding container becomes filled-up after some time, as the preceding peers are still writing to this container, but the writing peer is blocked. Eventually, from the container that was full, an Entry is taken. Thus, the blocked peer unblocks and writes its result to the container. It takes an Entry from its container, and the peers that were blocked because this container was full, are also unblocked and so on. So, if there is a blockage in a container, the blockage is accumulated to the preceding containers (see also "backpressure" in [6]). As an alternative, instead of letting the peer block,

we could let it detect the blocking via an exception (instead of using an infinite timeout) and let it write a Request for Help for the target container:

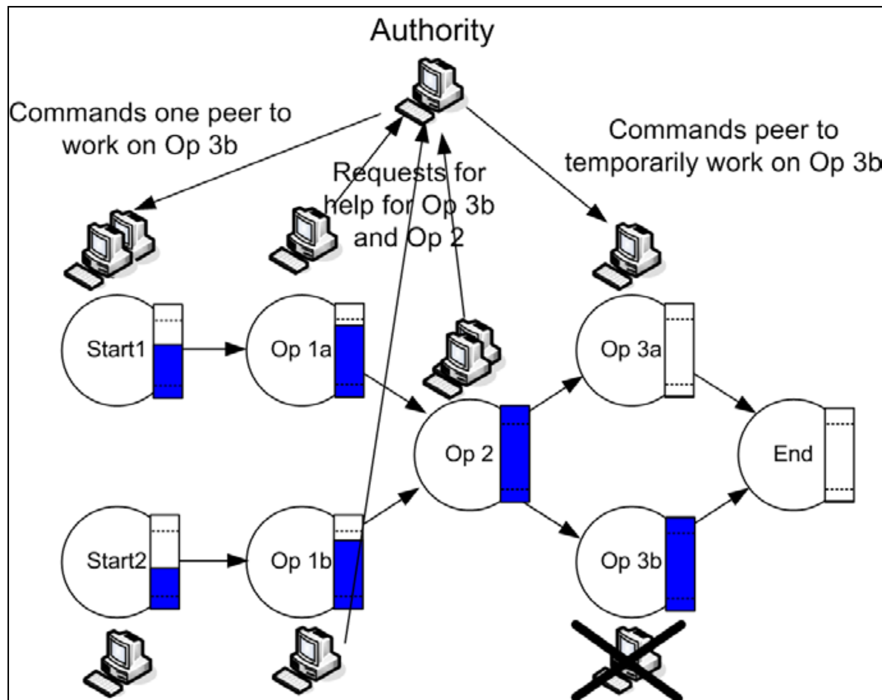


Figure 2.6.1 Accumulated blockage with Requests for help for the target container

As you can see, instead of the automatic load-balancing strategy explained in chapter 2.5, the Authority is used again, as it needs to decide what to do if the Resource Pool is empty. In this case, it needs to tell a specific peer to leave its container and work on the container that is full to resolve the blockage ("re-scheduling"). Remember that the peer that wants to **write** to the container which is full must contact the Authority, in contrast to the previous scenarios, where the peers that are noticing that the container they are currently **working** on, are contacting the Authority themselves.

This approach can be used in combination with the previously described automatic load balancing strategy: If the maximum threshold is reached on the own container, the peer writes a Request for Help to the Resource Pool. If a peer notes that the subsequent container is full, it writes a Request for Help for the subsequent container – remember that for the case the subsequent container's peer already has requested help, the request must remain unique in the Resource Pool, as already explained above.

If a single peer fails whereas there are others that are working on a container, it doesn't matter at all: The load of this container will most probably reach after some time the maximum threshold, a Request for Help is written and they will receive help, if available.

There are two problems unsolved:

If a peer fails while it already has taken an Entry from the container, the Entry is most probably lost. But the client that uses the execution path may be aware that probably none is working on its request and may take special actions to react on such situations. This is one of the advantages when working cooperatively instead of imperatively, that the Client is aware that the request wasn't successfully performed because the state is visible in the container. See also Chapter 1.4, about "Being more honest".

The Entry of the failed peer is not necessarily lost forever. Remind that the peer that is working on the current container might have a persistent container itself. Or it implements persistency by any other means, for example by writing the taken Entry to the file system. So whenever a peer takes an Entry from the container it is working on, it persists this Entry by some means. When the peer is restarted, before taking an Entry from the container, it checks whether there are Entries in its persistent storage and if so, these Entries are processed first.

If a peer that holds a container is stopped, all Entries of that container are lost. But if the peer that holds the container recovers, and if the container was a persistent one, all the Entries can be recovered.

Although these issues require some programming effort, compare this to the solution you have to design in the client-server architecture: you send the request to the server, if it fails, also here, your request and all its semi-results are lost. But using XVSM, you can at least recover the complete execution path after a failure of a single peer. In addition, patterns can be developed that support these recurring scenarios.

Conclusion

There are a lot of possibilities that XVSM can be used for. Of course message queues and topic queues are alternatives to XVSM, which you can use to implement any desired functionality, but XVSM has the advantage that it offers a high abstraction through arbitrary access to shared data containers that goes beyond directed first-in-first-out access, and that it is extensible, so simply through adding a new Aspect, you can let your peer use load balancing or you can let it work with high-priority requests. By using the cooperative way of telling another peer to perform an action, the peer could either be a single processor core, a mainframe, a cluster, it doesn't make any difference, as there is no distinction made, they are just referred as “peers” that do some action.

So XVSM provides (some parts are not implemented in the current release) to persist the data, spread it over multiple peers using replication techniques, perform requests with different priorities etc. This can help to easily implement scenarios that perform load-balancing in a path of executions, and/or support recovery by re-starting the peer and retrieving the data from the persistent storage. You can also be notified if certain data has arrived or was taken, or about execution of any other action in your execution path. With these properties, you can easily create a grid alike computing framework where the calculating entities can dynamically join or leave the network.

One main property of XVSM is that it is extensible: you can dynamically improve the functionality of your peers during runtime.

XVSM Tutorial

Version 1.8

Written by Michael Wittmann (xvsm_tutorial@gmx.at)

Examples and Exercises implemented by Laszlo Keszthelyi and Rene Formanek

Table of Contents

Chapter 1: Introduction 40
1.1 "Hello World!" – “Hello Space!” 42

Chapter 2: The Data structures 43
2.1 The Container, an introduction 43
2.2 Entry 47
2.3 Coordinating the contained things 48
2.4 Selectors 55
2.4.1 VectorSelector 56
2.4.2 KeySelector 57
2.5 Methods to access data 58
2.5.1 read 58
2.5.2 destroy 59
2.5.3 take 60
2.5.4 write 60
2.5.5 shift 61
2.6 Examples for Implicit Coordinators 61
2.6.1 Example: The Lottery (RandomCoordinator) 61
2.6.2 Exercise: The Ticket Queue (FIFOCoordinator) 62
2.7 Examples for Explicit Coordinators 63
2.7.1 Example: The Student management (KeyCoordinator) 63
2.7.2 Exercise: Formula 1 Race (VectorCoordinator) 64
Summary: 64

Chapter 3: Transactions 66
Exercise 3.1 (Transactions): 68
Summary: 69

Chapter 4: A completely new aspect: Aspects in XVSM 70
4.1 Local Aspects 70
4.2 Global Aspects 74
Summary: 75

Chapter 5: Don't miss a thing: Notifications 76
5.1 Exercise (TicketQueue with Notifications) 77
Summary: 78

APPENDIX: Solutions for the exercises 79
The TicketQueue (see Exercise 2.6.2): 79
Formula 1 Race (see Exercise 2.7.2): 80
Formula1Race extended by Transactions (see Exercise 3.1) 83
TicketQueue extended by Notifications (see Exercise 5.1) 90

Chapter 1: Introduction

XVSM is standing for eXtensible Virtual Shared Memory, which is middleware technology to store data objects in a space that can be shared with other peers. Data objects are written and read to that common data storage.

The space-based approach has some nice advantages. Many implementations of such an approach offer the possibility to distribute data over multiple computers, which can help to be more fault-tolerant, when one participating computer fails or leaves the space. In this case, the data is not necessarily lost as it may be stored on another computer. Moreover, when multiple computers can access the same data storage, they can cooperatively work on this data. For example, you can implement a system where a space participant that takes one entry of the space and perform some action on it, while another participant waits for the result of this calculation. The two participants can be separated from each other in physical space as well as in time. This means, that each participant can run on its own computer and not on the same one, and they don't need to run at the same time, because the calculation result remains in the data space after writing it. Thus, the space participant that waits for the calculation result can access this entry even after the participant that calculated it is no longer present. It is also possible to let multiple computers work on the same space to cooperatively and concurrently work on the set of data to improve the performance to handle the stored data objects.

XVSM in the current version offers the possibility to manage the data entries and the order and number of entries that are retrieved. For example, you can define a Coordinator to retrieve the entries in the opposite order they were written and another Coordinator to retrieve them in random order.

It also offers the possibility to perform actions within one transaction, this is to perform it as one atomic action that either executes successfully and thus the results are written to the space or when an error occurs, the whole transaction is rolled back and all changes are discarded. This reduces the danger that the data objects in the space are inconsistent after an error.

You also can register listeners on actions that are performed either on the data or on the space itself. If an action is performed that you are listening on, the corresponding

method to react on this event is called. This can be used to react for example when a data entry is added or removed. Such a technique can be used to enhance the functionality of the system, even while runtime, as these handlers can be registered and unregistered even while the system is running.

As a precondition to understand how to use XVSM and the terminology it is recommended that you first read the Application scenarios document, which gives an overview about various programming techniques in Space-Based computing and in XVSM in particular. The tutorial helps you to start programming using MozartSpaces, which is an open source implementation of XVSM in Java. It is not intended that this tutorial is a complete documentation of the API, it only gives you advice how to program with XVSM. You can find the complete API reference at [16].

In the next sections, an introduction to the Container and the ready-to-use data structures that can be used within the container is given. Then, this document tries to show you how to select specific data in the space. After that, it shows you how to extend the functionality of XVSM by using Aspects. As a usage of Aspects, the usage of notifications is explained, which are used to tell the listening program if new data is available, data has changed or data has been deleted.

Please note that XVSM is prepared for various additional functions, including the distribution of the data over multiple peers, automatic data persistency, and many more. This tutorial uses the most recent Java implementation of XVSM, which is MozartSpaces version 1.0, provided by the Space Based Computing Group [17] at the Technical University of Vienna. In this version, these functions are not yet implemented. Thus the tutorial focuses on the functions that are working in this version, so you can try out all examples and exercises that are given in this tutorial.

Further please note that MozartSpaces only works with Java 5.0 or newer.

1.1 "Hello World!" – “Hello Space!”

The "Hello World!" example is nearly mandatory for each new programming language or paradigm. Thus also here, an adaptation of this example is described. The details of it are explained in the subsequent chapters.

The purpose of this example is to show the basics of the new paradigm and to have a fast overview of its usage. In the original example, the text "Hello World!" shall be printed on the screen. In the XVSM example, the text “Hello Space!" is used and as a further variation this text shall be written to a shared container from which it is read and printed on the screen:

```
public static void main(String[] args)
{
	try
	{
		ICapi capi = new Capi();
		// Create new Container using no transaction, no remote server,

		// infinite container-size and no Coordinator
		ContainerRef cref = capi.createContainer(null, null, null,
			ICapi.INFINITE, null);

		/* Create a new AtomicEntry */
		Entry entry = new AtomicEntry<String>("Hello Space!");
		/* Write the entry to the container */
		capi.write(cref, 0, null, entry);

		// read and print the written value
		Entry[] readEntries = capi.read(cref, 0, null, null);
		System.out.println(((AtomicEntry) readEntries[0]).getValue());

		/* Shutdown and clear space */
		capi.shutdown(null, true);
	}
	catch (Exception e)
	{
		e.printStackTrace();
	}
}
```

The `ICapi` interface is the connection interface which offers you all methods to operate with MozartSpaces. For example, you use it to create or write to a container, or to access the Entries that are stored in an container. The details about containers and Entries are explained in Chapter 2.

When you have instantiated the `Capi` (Core Application Programming Interface) class, which is the implementation of the `ICapi` interface, you can create a container. Then you write an Entry containing the String "Hello Space!" to that container. Afterwards, this Entry is read again from the container and written to System.out.

Chapter 2:
The Data structures

There are several possibilities to read and write data from and to the XVSM space. As a precondition to understand what kind of data you'll be able to write to the space, it is necessary to show some basics about what the space actually is and how you can work with it.

2.1 The Container, an introduction

The Container is the place where the data is stored in. You can visualize the container in such a way:

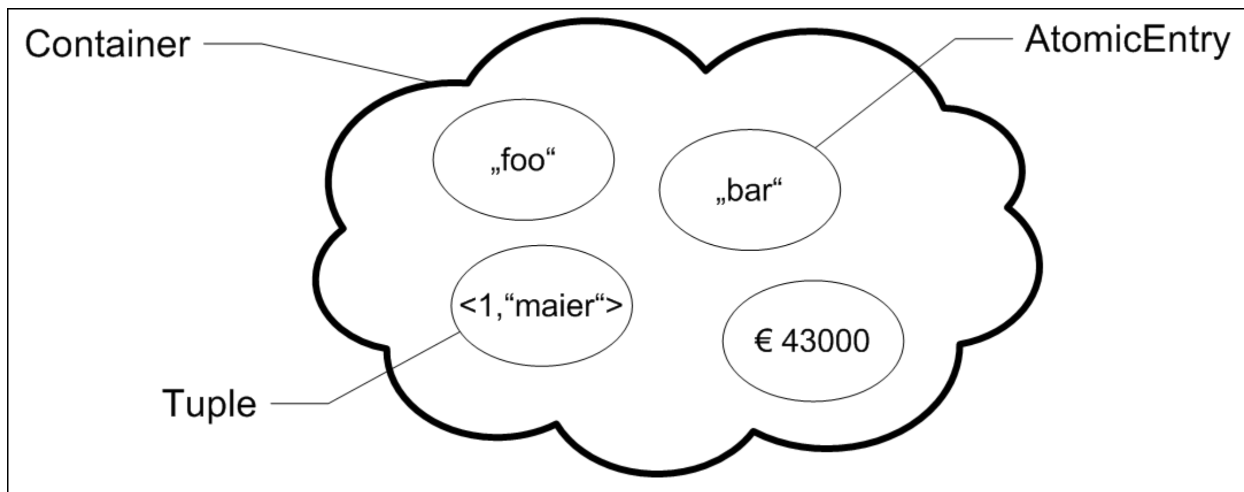
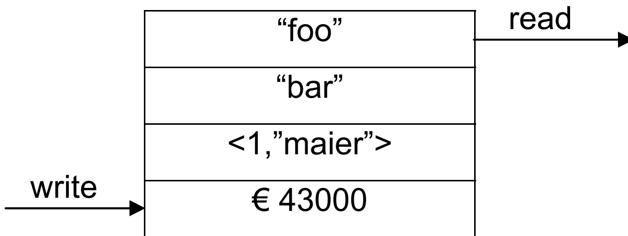


Figure 2.1: Overview of a container with objects

To give you additional information about the Entries, a table representation of the container is sometimes used. The arrows at the sides give the information of the order in which the Entries are written or read, which is especially useful when you use Implicit coordination (to be explained later). Such a table could look like this:



The data items that are stored in a container are called “Entries". An Entry can be either of type `Entry.EntryTypes.TUPLE` or `Entry.EntryTypes.ATOMICENTRY`. A Tuple contains other Entries, which can be either AtomicEntries or other Tuples. An AtomicEntry is a Generic Java class; when instantiating it, you can define the class that is contained within the AtomicEntry:

```
org.xvsm.core.Entry entry =
  new org.xvsm.core.AtomicEntry<String>("Hello World!");
```

A Container can either be located on your local machine or on a remote machine:

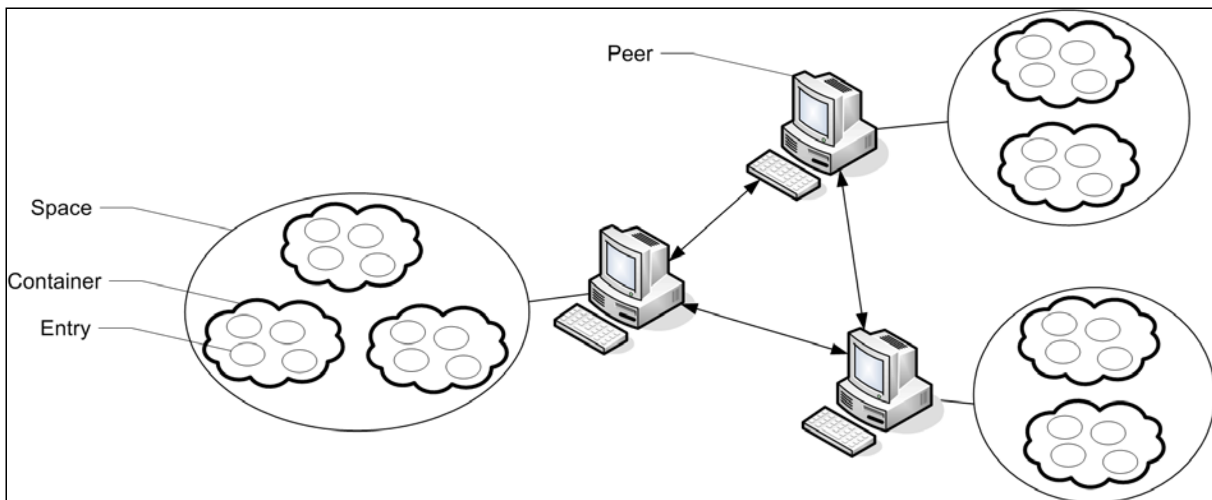


Figure 2.2: Communication of a set of peers, each with its own Space, Containers and Entries

The `Capi` class is an implementation of the `ICapi` interface, which is the main management class of the containers and XVSM itself. Using it, you can shutdown and restart the space, create and destroy containers, or read, write and destroy Entries in a container. First of all, you start your local space by calling

```
org.xvsm.core.ICapi capi = new Capi();
```

Every time new Capi() is called on a computer, a local Space is created which is accessible by other peers by default on port 1234. Another possibility to create a Space on a computer is to start the standalone Server, which you can find in the org.xvsm.server package and which has its own main method. It only starts a Space to handle the requests by other peers but doesn't offer an interface to handle the Space on its own, thus only other peers can manage the content of the Space. Keeping this in mind, there are two possibilities to start a new Space: by instantiating Capi or by starting a standalone Server.

To create a new container, use this method:

```
public ContainerRef createContainer(
              Transaction tx,
              URI site,
              int size,
              ICoordinator... coordinators)
        throws XCoreException
```

To create a container, you need to pass the following parameters:

- Transaction: transactions are explained in chapter 3. Use null to perform the action with an implicit transaction (this means that the action is automatically committed)
- URI of the Container's site: this is the URI to access spaces on a remote peer. By default, the port to connect to the remote peer is 1234, but of course the peer could use another port. The URI of a remote Server is constructed the following way: `new URI("tcpjava://mycomputer.mydomain.com:1234")`. The default port can be changed in the space.prop configuration file. Use null to connect to your local machine.
- Container's name. It can be used if you want to obtain a container after restart (only if persistency is already implemented by the XVSM version you use) or from another peer. You can use `null` to create an unnamed container. But doing this way is only preferable if you don't want to use that container after a restart of the space or if it is not necessary to share the space with other peers in a convenient way, as otherwise it would be complicated to access that unnamed container, as you need to use the automatically generated id as container name, which may not be an easy name to remember.

- Container's maximum size: The maximum number of entries that the container can hold. If you try to write an Entry to a container which already holds the maximum number of Entries, the write operation blocks. There also exists a shift method, which – instead of blocking – deletes “disturbing” Entries before writing the new ones. If you don’t want to have such a maximum size, use `IContainer.INFINITE_SIZE` as size parameter.
- List of Coordinators: You can define the internal coordination of the container. The ICoordinator interface is extended by the IImplicitCoordinator and IExplicitCoordinator interfaces. They are implemented by various classes, like the KeyCoordinator or FifoCoordinator. They are defining the sequence of the retrieved Entries when you call a method to read from the container. The different types of Coordinators will be described in Chapter 2.3. If you set the Coordination to null, the RandomCoordinator will be used, which retrieves the entries randomly.

The createContainer method returns a ContainerRef, which is in turn used to refer to this container. In the later examples, you will see that it makes sense to use multiple ContainerRefs. The ContainerRef is passed as parameter in most of the methods that are used to access data.

If you know that the container already exists or a `ContainerNameOccupiedException` (which extends `XCoreException`) is thrown on the creation, you can use the following method to get access to an existing (even remote) Container:

```
public ContainerRef lookupContainer(Transaction tx, URI site,
        String containerName) throws XCoreException
```

The parameters of this method are similar to the ones used in createContainer. This way, you can access a container which already exists on another peer.

Both createContainer and lookupContainer, as well as most other methods in MozartSpaces, can throw an XCoreException, which is the generalization of most XVSM-specific Exceptions and also stands for an Exception that is thrown because of an internal error.

When you'd like to shut down the Capi peer, you can pass the URI where the peer is running. Thus, you can smoothly turn off a remote peer, which makes sense especially for the standalone Server. The clearSpace flag is used to clean the space before you shut down the peer. This functionality is useless until persistency is supported, but for future implementation it already exists:

```
void shutdown(URI site, boolean clearSpace) throws XCoreException
```

Keep in mind that in the current implementation, no security mechanisms exist that keep a client away from restarting or clearing the space. Security mechanisms to prevent unauthorized users from performing these actions will be implemented in future releases of MozartSpaces.

Further note that currently, no persistency is implemented, so if you stop your peer, all information is lost, no matter if you set the clearSpace flag when shutting down the peer or not.

2.2 Entry

As already described, an Entry can be either a Tuple or an AtomicEntry. The AtomicEntry is a generic class, this means you can define what kind of data is stored in the entry:

```
Entry entry = new AtomicEntry<String>("Hello World!");
```

In the current version of MozartSpace, the class that can be used as type parameter in the AtomicEntry must implement the Serializable interface.
If you use the methods to access data, you'll see that in most cases you'll get an Entry returned, which is the superclass of Tuple and AtomicEntry, so you can't immediately get the value of the Entry. But Entry has a method getEntryType(), which returns one value of the EntryTypes enum that indicates what kind of Entry is received. Thus, using

```
if(readEntry.getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
```

you're able to cast the readEntry object to an AtomicEntry and then, you can access the contained data by calling its getValue method.
A Tuple entry implements the Iterator interface (Iterator<Entry>), thus it is ready-to-use to iterate over the entries that are contained within the tuple.

One major property of the Entry object is that when you instantiate it, you can pass a set of Selectors (which will be described in the next chapter):

```
protected Entry(Selector... sels)
```

So, for example

```
Entry entry = new AtomicEntry<String>(
        "writeCar",
        String.class,
        new VectorSelector(i, 0));
```

creates a new AtomicEntry of type String with the value "writeCar" and uses the VectorSelector to write it to a Vector at the position i.

Please take a look at the complete example at the end of this chapter to better understand how the Entries are created and used in the container.

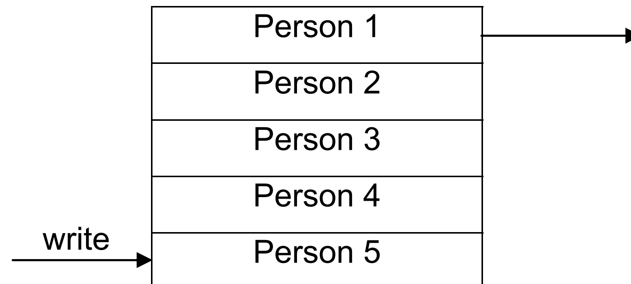
2.3 Coordinating the contained things

Coordinators are responsible for the order of the entries in the container that a programmer observes. You can access the Container either by using a Selector or without a Selector. A Selector is needed to refer to a specific Entry in the Container, like the Entry which holds the id=3. Generally speaking, Coordinators that implement the IImplicitCoordination interface have a view over all Entries in the Space, whereas those that implement the IExplicitCoordination interface have a view over all or a subset of the Entries, about which the Coordinator has stored additional info (like a key, an index,..). If you want to use the functionality of a specific Coordinator that you've specified when creating the container, you must pass the corresponding Selector when accessing Entries in this container (see also Chapter 2.4). If you leave the Selector away when accessing an existing Entry, the RandomCoordinator will be used. When writing to a container and you'd like to use the functionality of the implicit Coordinator, you don't need to pass it a Selector. In the current version, the following possibilities exist to coordinate the order of the retrieved Entries:

Implicit (complete view over all Entries):

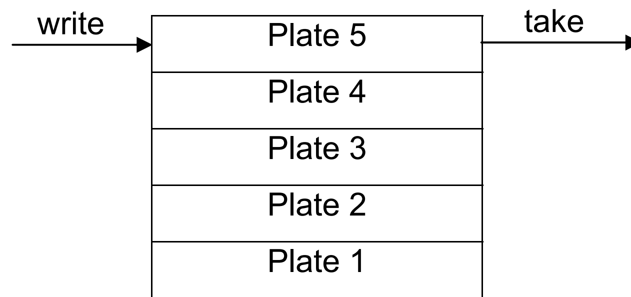
- **FifoCoordinator**: If you use the FifoCoordinator (FIFO stands for first-in-first-out), the entries are retrieved in the same order as you wrote them to the

container. This means, if you write 5 entries to the container, then you call the `read(cref, timeout, transaction, fifoSel)` method, the entry that you wrote FIRST is returned, thus the name "first-in-first-out". You can visualize this like a queue of people standing in front of the vendor's desk.




```
FifoSelector sel = new FifoSelector();

// Create new Container using no transaction, no remote server/peer,
// container-size of 5 and a FifoCoordinator
ContainerRef cref = capi.createContainer(null,
    null,
    null,
    5,
    new FifoCoordinator());

/* Fill the queue with the numbers 1 to 5*/
for (int i = 1; i <= 5; i++)
{
    Entry entry = new AtomicEntry<Integer>(i);
       capi.write(cref, 0, null, entry);
}
// read and print the first entry (1)
Entry[] readEntries = capi.read(cref, 0, null, sel);
System.out.println(((AtomicEntry) readEntries[0]).getValue());
```
Listing 2.3.1: FifoCoordinator

- **LifoCoordinator**: If the container uses the LifoCoordinator (LIFO stands for last-in-first-out), the entries are retrieved in the reverse order you wrote them. Thus, if you write 5 entries and then call the `read(cref, timeout, transaction, lifoSel)` method, the entry that was written LAST will be returned. You can compare this to a stack of plates – The last one you put on the stack will be the first one that you take from that stack.




```
LifoSelector sel = new LifoSelector();

// Create new Container using no transaction, no remote server, container-
// size of 5 and a LifoCoordinator
ContainerRef cref = capi.createContainer(null,
    null,
    null,
    5,
    new LifoCoordinator());

/* Fill the queue with the numbers 1 to 5*/
for (int i = 1; i <= 5; i++)
{
    Entry entry = new AtomicEntry<Integer>(i);
    capi.write(cref, 0, null, entry);
}
// read and print the last value (5)
Entry[] readEntries = capi.read(cref, 0, null, sel);
System.out.println(((AtomicEntry) readEntries[0]).getValue());
```
Listing 2.3.2: LifoCoordinator

- **LindaCoordinator**: This is a Coordinator that retrieves data using Linda template matching [18]. You write the Entries (which must be tuples for this kind of coordination) to the Container; when you want to retrieve specific data, you define a template Entry. This template is a standard Entry, as already used when writing the data. For example, you write the following tuples to the Container:

  ```
  <<"hello", "world">, 1>
  <<"hello", "world">, 2>
  <<"hello", "world">, 3>
  <<"hello", "world">, 4>
  <<"hello", "world">, 5>
  ```

  Now you define a template Entry, which looks like:

  ```
  <<"hello", "world">, null>
  ```

  Now, all entries are retrieved.

  If you define as template this Entry:

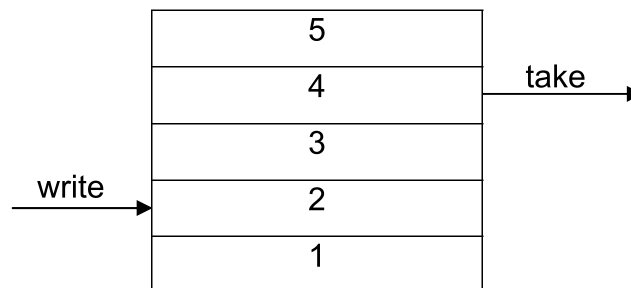
  ```
  <<"hello", "world">, 1>
  ```

  then only one Entry will be retrieved, as there is only one Entry in the container that matches this template. Thus, the template Entry needs to have the same structure (in terms of the Tuple and AtomicEntry structure). You define the fields of the Entry where you require exact matching and null where you don't require exact matching of the Entry.

```
//Create new Container using no transaction, no remote server,
//container-size of 5 and a LindaCoordinator
ContainerRef cref = capi.createContainer(null, null, null, 5,
                           new LindaCoordinator());

//Fill the container with <<"hello","world">,i> tuples where
//i = 1 to 5
for (int i = 1; i <= 5; i++)
{
Entry intEntry = new AtomicEntry<Integer>(i);
Entry entry = new Tuple(new Tuple(
                           new AtomicEntry<String>("hello"),
                           new AtomicEntry<String>("world")
                           ),new AtomicEntry<Integer>(i));
capi.write(cref, 0, null, entry);
}

//this template retrieves all entries as the last field in the tuple
//is null, but it is the only difference between the written tuples
Entry template1 = new Tuple(new Tuple(
                           new AtomicEntry<String>("hello"),
                           new AtomicEntry<String>("world")
                           ), null);
LindaSelector sel1 = new LindaSelector(Selector.CNT_ALL, template1);
```

```
Entry[] readEntries1 = capi.read(cref, 0, null, sel1);
printAllEntries(readEntries1);

//this template retrieves only the Entry with the 1 as last field in
//the tuple
Entry template2 = new Tuple(new Tuple(
                           new AtomicEntry<String>("hello"),
                           null
                           ),new AtomicEntry<Integer>(1));

LindaSelector sel2 = new LindaSelector(Selector.CNT_ALL, template2);

Entry[] readEntries2 = capi.read(cref, 0, null, sel2);
printAllEntries(readEntries2);
```
Listing 2.3.3 LindaCoordinator. The printAllEntries method is not listed here – it only prints the Entries' content one after the other.

- **RandomCoordinator**: This is the standard Coordinator (if you don’t use a Coordinator at all). The entries are returned in a random way. It is used by default when you don't pass a Selector for reading, deleting etc. of entries (see Chapter 2.4). You can compare this to a bag with lottery numbers:



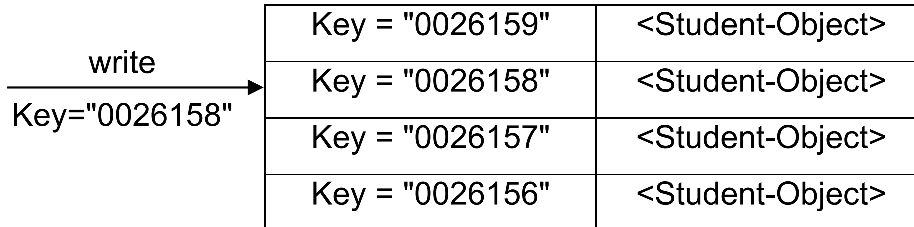

```
ContainerRef cref = capi.createContainer(null, null, null, 5, null);

/* Fill the queue with the numbers 1 to 5*/
for (int i = 1; i <= 5; i++)
{
    Entry entry = new AtomicEntry<Integer>(i);
    capi.write(cref, 0, null, entry);
}
// read and print a random value (something from 1 to 5)
Entry[] readEntries = capi.read(cref, 0, null, null);
System.out.println(((AtomicEntry) readEntries[0]).getValue());
```
Listing 2.3.4: RandomCoordinator (default)

Explicit (you need to give extra information when writing or reading the Entry):

- **KeyCoordinator**: When you use the KeyCoordinator, you can define a key when you write an entry. When you want to find this very entry again, you use the same key to get exactly this entry. So, for example, you have an AtomicEntry which holds an object of type Student, which has several properties. When you write the entry to the container, you use the Student’s

take

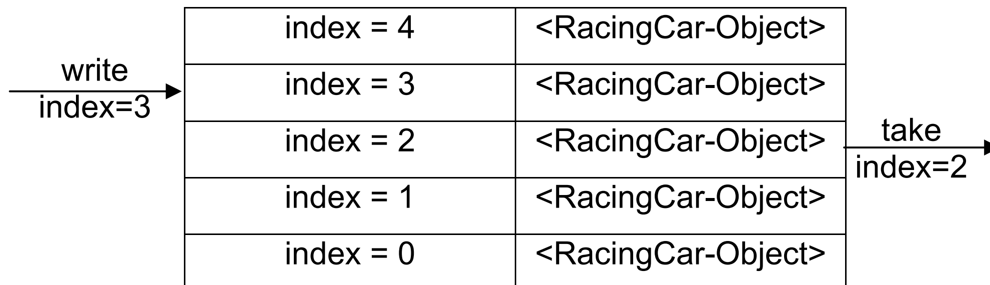
matriculation number as key. When you later want to find this Student in the container, you need to use the KeySelector together with the Student's matriculation number, which is the key.



The name of the key must be a String, the key itself is generic, enabling to use any Java object, which is in turn passed as the type parameter.

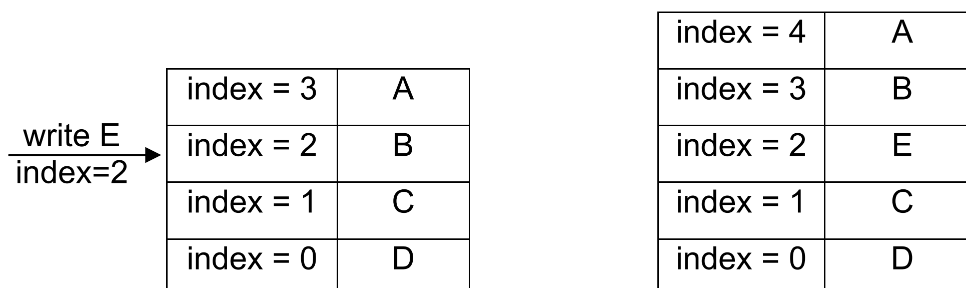
```
/* Create new Container using no transaction, no name, an infinite */
/* container-size and KeyCoordinator */
ContainerRef cref = capi.createContainer(null, null, null,
    IContainer.INFINITE_SIZE,
    new KeyCoordinator());

/* Create a Student instance and write it into the container */
Student writeStudent = new Student(1000, "Max", "Muster", 20);

/* Create new AtomicEntry using KeySelector */
Entry entry = new AtomicEntry<Student>(writeStudent, Student.class,
    new KeySelector<String>("Surname", writeStudent.get_Surname()));
capi.write(cref, 0, null, entry);

/* Get entry using KeySelector from the container */
Entry[] readEntries = capi.read(cref, 0, null,
    new KeySelector<String>("Surname", "Muster"));

if (readEntries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
{
    Student readStudent = ((AtomicEntry<Student>) readEntries[0]).getValue();
    System.out.println("MatNr.:    " + readStudent.get_MatNr());
    System.out.println("Forename:  " + readStudent.get_Forename());
    System.out.println("Surname:   " + readStudent.get_Surname());
    System.out.println("Age:       " + readStudent.get_Age());
}
```
Listing 2.3.5: KeyCoordinator

- **VectorCoordinator**: The VectorCoordinator is used when you want to address the Entries by index. You can compare this with the position of Racing cars.

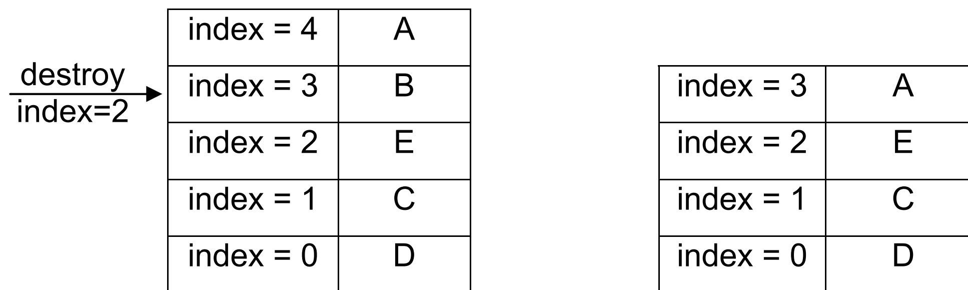



```
ContainerRef cref = capi.createContainer(null, null, null,
    IContainer.INFINITE_SIZE,
    new VectorCoordinator());

/* Create 5 Strings and write them to the container  */
for (int i = 1; i <= 5 ; i++)
{
    Entry entry = new AtomicEntry<String>(i+". Person", String.class,
             new VectorSelector(VectorSelector.APPEND, 0));
    capi.write(cref, 0, null, entry);
}
/* Get entry using VectorSelector from the container */
Entry[] readEntries = capi.read(cref, 0, null, new VectorSelector(2, 1));

if (readEntries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
{
    System.out.println(((AtomicEntry<String>) readEntries[0]).getValue());
}
```
Listing 2.3.5: VectorCoordinator

One property of the VectorCoordinator which might be a bit tricky for beginners is that when you write an Entry to an index where another Entry already resides, this existing Entry is moved by one (its index is automatically incremented). All Entries' subsequent indexes are also incremented by 1. You can imagine this like pushing all Entries with higher indexes one position further:



Note that B was stored with index 2, but now it is stored with index 3. On the other hand, when you destroy an Entry from a container that is coordinated by a VectorCoordinator, the indexes of the Entries that have a higher index than the destroyed one are decremented by one:



The Entry A was stored with index 4, after the destroy command, it was moved to index 3.

Thus you need to keep in mind that the indexes of Entries may be changed when adding/removing Entries.

2.4 Selectors

The Selector is the counterpart to the Coordinator. It tells additional information to the Coordinator when you want to read or delete data, like the key value that you're searching for or the number of values that you'd like to retrieve. For each Coordination type, there exists the appropriate Selector.

The Selector class is abstract and offers two constructors:

```
public Selector()
public Selector(int count)
```

The count parameter is the number of entries that you'd like to retrieve using this Selector. If you leave away the count parameter, only one Entry is returned; when using the KeySelector, the Entry that has this key is returned. If you would like to have all entries of the container that fulfil the selection, use `Selector.CNT_ALL` as count parameter. Note that in the Linda example in chapter 2.3, I assumed the usage of the Selector.CNT_ALL parameter. When you use the LindaSelector without a count parameter, only one Entry is returned, if a matching one exists.

If you create an Entry and write it to the container, you may pass a Selector as parameter (see Chapter 2.2). As Implicit Coordinators have the view over all Entries in the container, you don't need to pass it explicitly when writing the Entry to the container – the Implicit Coordinator manages the Entry anyway. When you pass an Explicit Selector together with its required additional info, the Explicit Coordinator

(which has a view over all or a subset of Entries in the complete container) will manage this Entry, together with the Implicit Coordinator (as it has the view over ALL Entries, even those that are managed by an Explicit Coordinator). Thus, for example, if you created the container and passed it a FifoCoordinator and a KeyCoordinator, then you can write an Entry with a KeySelector, but still will be able to retrieve it using the FifoSelector.

The currently implemented implicit selectors only offer the constructors mentioned above, as the sequence of the retrieved entries is given implicitly (remember, the implicit selectors currently are: FIFO, LIFO, Linda, Random).
The explicit selectors on the other hand offer a bit more information to the Coordinator, therefore, they are explained separately.

2.4.1 VectorSelector

```
public VectorSelector(int index, int count)
public VectorSelector(int count)
```

The VectorSelector has the additional parameter index, which indicates the number of Entries in the container that you'd like to access. Note that if you write an Entry to the container, the numbers of the other entries with higher and equal numbers are shifted by one, which means that each index of an Entry that has a higher index is incremented by 1.
For example:

```
Entry entry = new AtomicEntry<String>(
  "The first one",
  String.class,
  new VectorSelector(1, 0));
```

2.4.2 KeySelector

```
public class KeySelector<T> extends Selector {
  public KeySelector(String keyName, T keyValue) ...
```

The KeySelector is a generic class, you need to define which object type the key will be. The keyValue parameter will be the same type that you defined when you created the KeySelector instance:

```
KeySelector<String> keySel = new KeySelector<String>("Name", "Maier");
```

Using this KeySelector, you could pass it as parameter to the Entry when you write it:

```
Entry entry = new AtomicEntry<Account>(
  new Account(10000),
  Account.class,
  keySel);
```

or you can use it when you want to retrieve this entry:

```
Entry[] readEntries = capi.read(cref, 0, null, keySel);
```

Note that the KeySelector doesn’t have a parameter for the count, as the key is unique and thus, either only one or no entry can be retrieved with this key. If you want to write an Entry to the container with a key that already exists, an exception is thrown. Please further note that the shift method doesn't throw this exception, as it first would remove the clashing Entry.

Please take a look at the complete example at the end of this chapter to better understand how to use the KeySelector.

2.5 Methods to access data

In this chapter, the used ContainerRef objects are the same instances that you got when you created the container. For now, ignore the Transaction parameter, when programming, you can use null instead of the tx. Transactions will be explained in chapter 3.

2.5.1 read

```
public Entry[] read(ContainerRef cref,
       long timeout,
       Transaction tx,
       Selector... sel)
    throws XCoreException
```

The read method returns a number of Entries from the container, without destroying them. You need to use a Selector that is corresponding to the Coordination. If you leave away the Selector parameter, RandomSelector is used. The implicit as well as the explicit Coordinators can have a count parameter in the constructor, which tells the container the number of entries that you'd like to retrieve (see chapter 2.4 about Selectors). Thus, the return value of the read is an array of Entries, because you can define a Selector to read and return multiple Entries at once. If you don't use the parameter to tell the number of entries that should be read, you get one entry (if there is at least one in the container) or none.

One important feature that can be used for synchronization is the fact, that if there is currently no entry in the container that can be retrieved, the read method blocks and waits for an Entry. And that's the reason why there is a timeout parameter: It tells the read method to wait at least the timeout in seconds. So giving a timeout of Timeout.NO_TIMEOUT, you have a non-blocking read method call – it either reads an entry or if there is none, it immediately returns an Exception. The opposite would be to use the Timeout.INFINITE_TIMEOUT, which waits forever, if no one ever writes a suitable entry to the container.

```
//Create new Container using no transaction, container-size of 5 and a
//FifoCoordinator
ContainerRef cref = capi.createContainer(null, null, null,
    5, new FifoCoordinator());

/* Fill the queue with the numbers 1 to 5 */
for (int i = 1; i <= 5; i++)
{
    Entry entry = new AtomicEntry<Integer>(i);
    capi.write(cref, 0, null, entry);
}
Entry[] readEntries = capi.read(cref, 0, null, new FifoSelector());
/* ... prints 1 */
System.out.println(((AtomicEntry<Integer>) readEntries[0]).getValue());

/* Read from the container with FifoSelector of size 2 */
readEntries = capi.read(cref, 0, null, new FifoSelector(2));
/* ... prints 1 */
System.out.println(((AtomicEntry<Integer>) readEntries[0]).getValue());
/* ... prints 2 */
System.out.println(((AtomicEntry<Integer>) readEntries[1]).getValue());

/* Read from the container with timeout */
readEntries = capi.read(cref, 5, null, new FifoSelector());
/* ... prints 1 */
System.out.println(((AtomicEntry<Integer>) readEntries[0]).getValue());
```

Listing 2.5.1: Reading entries using a FifoSelector. One time with size = 2 and the second time without size given. Also one read with a Timeout given and one without. As transaction, null is used.

2.5.2 destroy

```
public void destroy(ContainerRef cref,
        long timeout,
        Transaction tx,
        Selector... sel)
    throws XCoreException
```

The destroy method adheres to the same rules as the read method. Depending on the container's Coordination type, an entry is destroyed, but without returning the destroyed entry. Note that also, when destroying entries, the method blocks until a suitable entry is found in the container. One might think that it is not very meaningful to wait for an entry just to immediately destroy it, but even though there are better techniques to perform this task, one might use this technique to notify a single listener. The listener calls `destroy(cref, Timeout.INFINITE_TIMEOUT)` and thus is blocked. The notifier writes an entry to the container and doing so, the entry is destroyed and the listener is unblocked. As already said, there are better possibilities in XVSM to perform such tasks – in most cases, you'll have entries when you call destroy, so this method won't block.

2.5.3 take

```
public Entry[] take(ContainerRef cref,
       long timeout,
       Transaction tx,
       Selector... sel)
    throws XCoreException
```

This method is the same as first calling read and then destroy: The read Entry is immediately destroyed after reading in one atomic step. This functionality is also commonly referred to as “consuming read”. Also the take method could block and thus has a timeout parameter.

2.5.4 write

```
public void write(ContainerRef cref,
       long timeout,
       Transaction tx,
       Entry... entries)
    throws XCoreException
```

By using this method, you can write one or more entries to the space. Also here, you use the ContainerRef object to refer to the corresponding container. When the container’s size is limited, and there are already as much entries in the container as allowed (the number of entries in the container is equal to the maximum capacity), and you want to write further entries to the container, this operation blocks. The write operation waits until an entry is taken or destroyed or – if given – until the timeout is reached. If you'd like to use a Selector together with a specific Entry (e.g. a KeySelector), this Selector already is passed when you created the Entry (see Chapter 2.4)

2.5.5 shift

```
public void shift(ContainerRef cref,
      Transaction tx,
      Entry... entries)
   throws XCoreException
```

The shift method has the same behaviour as the write method, but instead of blocking if the container is already full, the shift method destroys an entry automatically. Thus, no timeout parameter is needed – it just never blocks. The entries that are shifted out of the container depend on the Coordinators of the Container and the Selectors used in the passed Entries. The shift method decides what Entry to destroy, in order to be able to write the new Entry.

2.6 Examples for Implicit Coordinators

2.6.1 Example: The Lottery (RandomCoordinator)

In this example, we would like to show how an Austrian Lottery[1] (in German it is called: “Lotto 6 aus 45”) can be programmed. As we assume that the lottery really picks the balls randomly, we take the RandomCoordinator with a size of 45.
First, we write the 45 numbered balls to the container. Then we take the 6 + 1 balls out of the container. At the end, we list the taken balls.

```
public class Lottery
{
   public static void main(String[] args)
   {
      try
      {
         ICapi capi = new Capi();
         Selector sel = new RandomSelector();

         /*
          * Create new Container using RandomCoordinator, RandomSelector and a
          * container-size limit of 45
          */
         ContainerRef cref = capi.createContainer(null,
            null,
            null,
            45,
            new RandomCoordinator());

         /* Write the 45 numbers to the container */
```

[1] for non-Austrians: you have 45 numbered balls, the moderator picks 6 out of them + 1 additional. If you have guessed the 6 correctly, you’ve won the major amount; also you get quite a lot of money if you have 5 correct + the 1 additional.

```
        for (int i = 1; i < 46; i++)
        {
           Entry entry = new AtomicEntry<Integer>(i);
           capi.write(cref, 0, null, entry);
        }

        /* Take 6 + 1 numbers from the container */
        System.out.print("Regular Numbers: ");
        for (int j = 0; j < 7; j++)
        {
           /* Take an entry from the container using the RandomSelector */
           Entry[] readEntries = capi.take(cref, 0, null, sel);

           if (readEntries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
           {
              System.out.print(((AtomicEntry<Integer>) readEntries[0]).getValue());
              if (j == 5)
              {
                     System.out.print("\nBonus: ");
              }
              else if (j < 6)
              {
                     System.out.print(", ");
              }
           }
        }
        System.out.println("\n\nHope you have better luck next time! ^_~\nbye!");
        /* Shutdown and clear space */
        capi.shutdown(null,true);

     }
     catch (Exception e)
     {
        e.printStackTrace();
     }
  }
}
```

Listing 2.6.1: The Austrian Lottery

2.6.2 Exercise: The Ticket Queue (FIFOCoordinator)

A bunch of people is standing in a queue to buy cinema tickets. One person after the other is served by the shop assistant. A person who wants to buy a ticket must pay and wait for the ticket to be printed. This takes some time (we assume that this is a quite fast action and takes 2 seconds in total).

Please program a Capi peer that handles the persons in the queue one after the other. As person objects, use an extra class “Person” that has a payTicket() and a waitForTicket() method. Both methods only wait for 1 second. In later examples, this exercise will be enhanced. The shop assistant only gets the first person in the queue, receives the payment and prints the ticket. Afterwards, the person object in the queue leaves the queue. You don’t need to have a shop assistant class, just write it in the main method of the main class.

2.7 Examples for Explicit Coordinators

2.7.1 Example: The Student management (KeyCoordinator)

The Technical University of Vienna would like to have a new student management system developed using XVSM. Each Student has a matriculation number (MatrNr), so we'll use a KeyCoordination, where the key is the MatrNr. As a quick test, we'll write 5 Student objects and read them by retrieving them by their MatrNr.

```
public class StudentManagement
{
   public static void main(String[] args)
   {
      try
      {
         ICapi capi = new Capi();
         /* Create new Container using KeyCoordinator and KeySelector */
         ContainerRef cref = capi.createContainer(null,
            null,
            null,
            IContainer.INFINITE_SIZE,
            new KeyCoordinator());

         Student students[] =
            {new Student(1000, "Max", "Muster", 20),
            new Student(1001, "Ernst", "Müller", 28),
            new Student(1002, "Nora", "Maier", 19),
            new Student(1003, "Indiana", "Jones", 26),
            new Student(1004, "John", "Constantine", 24)};

         /* Create 5 Student instances and write them into the container */
         for (Student writeStudent : students)
         {
            /* Create new AtomicEntry using KeySelector */
            Entry entry = new AtomicEntry<Student>(writeStudent,
               Student.class,
               new KeySelector<Integer>("MatrNr", writeStudent.getMatNr()));
            capi.write(cref, 0, null, entry);
         }

         /* Read entries */
         for (int i = 1000; i < 1005; i++)
         {
            /* Get entry using KeySelector from the container */
            Entry[] readEntries = capi.read(cref,
               0, null,
               new KeySelector<Integer>("MatrNr", i));

            if (readEntries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
            {
               Student readStudent = ((AtomicEntry<Student>)
readEntries[0]).getValue();
               System.out.println("*** Beginning of entry ***");
               System.out.println("MatNr.:    " + readStudent.getMatNr());
               System.out.println("Forename: " + readStudent.getForename());
               System.out.println("Surname:  " + readStudent.getSurname());
               System.out.println("Age:      " + readStudent.getAge());
               System.out.println("*** End of entry ***\n");
            }
         }
```

```
         /* Shutdown and clear space */
         capi.shutdown(null,true);
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}
```
Listing 2.7.1: Student management

2.7.2 Exercise: Formula 1 Race (VectorCoordinator)

You're the IT professional in managing the Race results of a Formula 1 Race. You have the task to implement a piece of software (using XVSM of course!) to show the current race position. The values of the entries are RacingCar objects, whereas they have the property "driverName". When someone accesses the container to retrieve the current status of the race, of course, all cars should be listed together with their current position. For simplicity, just write 10 cars to the container. In the first round, car number 6 crashes and thus is removed from the race. In the second round, car 4 overtakes car 3. As this is a short race, let's say the race is over after this lap. List the current racing positions at the end of each lap. Also this example will be improved later. Each Racing car is running on a separate Peer (thus, one peer creates and holds the container, whereas all the other peers are connecting and using this container then).

Summary:

In this chapter, you've learnt the most important basics to handle a container and a container's content. The basic access to the container is done using the Capi class, which offers methods to create, lookup and destroy a container and to read, write, destroy and shift entries from/to the container.
An entry can be either a tuple or an atomic entry. A tuple can hold either other tuples or atomic entries. An atomic entry is a Generic that holds the data of interest.
You also learnt how to set the internal coordination of the container and how to select specific data in the container.
A typical use case scenario is that you'd like to create a new container and pass it the internal coordination type. Then, you'd like to create some entries and set the information in the entries. Afterwards, you'll write the entries to the container and

then read or take them from the container. At the end, most probably you’ll shutdown the Capi to unlink from the container.

Chapter 3: Transactions

We assume that most programmers are familiar with the concept of transactions: A transaction is helping to collect various actions to perform them as one single, atomic step to avoid consistency problems of the data storage. In the XVSM implementation, the transaction is performed as a pessimistic transaction; this means a transaction builds up locks on the data.

The lifecycle of a transaction from the user's perspective is as follows:

- Create a new transaction
- Perform various actions
- If everything went fine, commit the transaction to write the changes to the data storage
- If there was an error, rollback the changes to discard all actions you made.

In XVSM, if you start a transaction and then only perform read operations, a read lock is obtained. But as soon as you perform a write operation (or another operation that changes the container's content), the container or the accessed Entries (depending on the isolation level implemented by the MozartSpaces version you use) in the container are locked exclusively for this one transaction. There are multiple read locks allowed, but only one write lock at a time. All other applications or application parts that would like to access the container but don't use this transaction are blocked, no matter if they want to read or write to that container, until the transaction is committed or rolled back. It depends on the used Coordinator, if the complete container must be locked or if it is sufficient to only lock the accessed Entries. For example, the FifoCoordinator will lock the complete container, because the sequence the Entries will be returned, must remain the same while the transaction, whereas using a RandomCoordinator, the locks will be obtained only on the accessed Entries.

You create a transaction by calling

```
ICapi capi = new Capi();
```

```
Transaction tx = capi.createTransaction((URI)site, (long)timeout);
```

As you remember from the previous chapter, this Transaction object is passed as parameter in various operations (read, destroy, write, …). So, if you call the write operation with the Transaction tx, the container resp. the corresponding part of the container is locked only for this transaction. Concurrent access to the container that is in conflict with the write operation will be blocked. Only the methods that use this tx transaction are executed without blocking. One thing that you need to keep in mind: if you don't set a transaction (you use null as parameter), internally, an implicit transaction is created. This means, internally, a transaction object is created – thus, also here, concurrent actions run isolated to each other, although the action is committed automatically.
The site parameter is the URI, where the container is located at (see Chapter 2.1) and the timeout parameter can give a maximum validity time of the Transaction. If the given time has elapsed, all changes within this Transaction are rolled back and all further actions with this Transaction are rejected.

You commit the transaction by calling:

```
capi.commitTransaction(tx);
```

This means all your changes are written to the container and the container is unlocked then. The transaction object no longer is valid then.

You roll the changes in this transaction back by calling:

```
capi.rollbackTransaction(tx);
```

This means that the actions that you've done so far are discarded. A rollback might be intended if some error occurred, so most probably, the rollback statement will be standing in an error handling part.

```
ICapi capi = new Capi();
tx = capi.createTransaction(null, Capi.INFINITE_TIMEOUT);
tx2 = capi.createTransaction(null, Capi.INFINITE_TIMEOUT);
try {

    /* Create new Container using a container-size of 3 */
    ContainerRef cref = capi.createContainer(null, null, null, 3,
            new FifoCoordinator());

    /* Write 3 entries to the container */
    /* After the first write action the container is locked */
    for (int i = 1; i <= 3; i++) {
        Entry entry = new AtomicEntry<Integer>(i);
```

```
        capi.write(cref, 0, tx, entry);
    }
    capi.commitTransaction(tx);
    /* The container is not locked anymore */

    capi.read(cref, 0, tx2, new RandomSelector(3));
    /* The container is still not locked */
    capi.destroy(cref, 0, tx2, new RandomSelector());
    /* The container is locked now */

    throw new Exception();

} catch (Exception e) {
    try {
        if (tx != null){
            capi.rollbackTransaction(tx2);
        }
        /* The container is not locked */
```
Listing 3.1: Transaction handling

Exercise 3.1 (Transactions):

Extend the exercise 2.7.2 (Formula 1):

We'd like to let some cars run for a random amount of time. So we'd start a Capi peer for each car, let it wait for some time and then re-register in the container within a transaction to have a consistent container content. As this is a bit tricky, we'll give you a guide how to solve this exercise:

1. Add the properties "runtime" and "lapNumber" to each car.
2. For each car, start a new Capi peer. When having started it, let the peer wait for a random time between 3 and 5 seconds:
   ```
   Random rnd = new Random();
   int time = rnd.nextInt(2) + 3; //(random between 0 and 2) plus 3
   ```
3. If the Time is up, start a transaction.
4. Add the time that you've waited to your runtime.
5. Remove the car that actually has the transaction from the container.
6. Read all actual positions of the cars, and get the runtime for each car. If you've found a runtime that is higher, whereas the next runtime is lower than your runtime, then this is your position.
7. Write the car object to the new position (remember that the entries in the vector with higher index than the one that you use will be shifted down automatically). Don't forget that the car only needs to take 2 rounds, but of course you can use more rounds – you just need to keep the number of rounds for each car the same ☺ …

8. Every time a car reaches the finish line, write the actual list of race positions together with the runtimes. Don't forget that cars that already have passed the finish line don't change their racing time any more.

Summary:

Transactions are fundamental if there are many peers (or threads) that use containers to avoid that through concurrent write access, the content may become inconsistent. A transaction helps you to keep the container consistent, as it serializes the transactions as if one were executed after the other. So normally, you'd like to start a transaction, read or write from/to the container and commit. All other application parts that want to access that part of the container but don't use this transaction are blocked.

Chapter 4:
A completely new aspect: Aspects in XVSM

An Aspect in XVSM lets you implement an extension to the existing XVSM functionality. Features like automatic persistency or notification can be implemented using Aspects. You also can use Aspects to add logging, authentication and many more features to the existing XVSM implementation. The idea behind Aspects is to have various pre- and post-methods on each action that the Capi class is able to perform, like a preWrite and a postWrite method. You implement these pre- and post-methods to perform some special action that is done automatically before or after you call one of the following methods of the ICapi interface. You need to divide Aspects into two groups: Local Aspects and Global Aspects. Local Aspects are aspects that are connected to actions on a specific container, while Global Aspects refer to the space itself or to all containers.
Each possibility to listen on a certain performed action is called IPoint (standing for Interceptor Points). Adhering to the just mentioned Local and Global Aspects, the corresponding IPoints are called `LocalIPoint` and `GlobalIPoint`.
You not only have the possibility to create one Aspect to perform an action, but you can also add multiple Aspects to perform various actions in sequence. This possibility will be explained at the end of Chapter 4.1.

4.1 Local Aspects

Local Aspects offer the possibility to overwrite the pre- and post-Methods for the following actions:

- addAspect()
- removeAspect()
- read()
- destroy()
- take()
- write()
- shift()

If you want to create a new local aspect, you have to extend the abstract LocalAspect class, if you want to create a new global aspect, you have to extend the abstract GlobalAspect class. They already contain the needed pre- and post-methods that per-se don't do anything. If you implement a new Aspect, you override these methods to add new functionality to the method.

An example for a new Local Aspect is:

```
public class LoggingAspect extends LocalAspect
{
      static final long serialVersionUID = 0;
      public void postWrite(ContainerRef cref,
                              Transaction tx,
                              List<Entry> entries, Properties p)
      throws AspectNotOkException, AspectRescheduleException,
                        AspectSkipException
      {
            try
            {
                  FileOutputStream fs = new FileOutputStream("Logfile.log",
true);
                  ObjectOutputStream os = new ObjectOutputStream(fs);
                  Iterator it = entries.iterator();
                  while (it.hasNext())
                  {
                        /* Write the value of the AtomicEntry to the
logfile */
                        os.writeObject(((AtomicEntry)
it.next()).getValue());
                  }
                  os.close();
            }
            catch (Exception e)
            {
                  e.printStackTrace();
            }
      }

      public void postRead(ContainerRef cref,
                                          Transaction tx,
                                          List<Entry> entries,
                                          List<Selector> selectors,
Properties p)
      throws AspectNotOkException, AspectRescheduleException,
                  AspectSkipException
      {
            try
            {
                  FileOutputStream fs = new FileOutputStream("Logfile.log",
true);
                  ObjectOutputStream os = new ObjectOutputStream(fs);
                  Iterator it = entries.iterator();
                  while (it.hasNext())
                  {
                        /* Write the value of the AtomicEntry to the
logfile */
                        os.writeObject(((AtomicEntry)
it.next()).getValue());
```

```
                }
                os.close();
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
}
```

Listing 4.1: Logging Aspect

In the listing above, you extend the LocalAspect class and override the various methods and therefore add new functionality. The methods could look like this:

```
public void postWrite(
                ContainerRef cref,
                Transaction tx,
                List<Entry> entries,
                Properties contextProperties)
    throws AspectNotOkException,
    AspectRescheduleException,
    AspectSkipException
```

The cref and tx parameters should already be clear. You also can give a list of Entries that are used in this action, to perform some special task with these entries, as you can see in the listing. The contextProperties are free to your disposal, if you want to pass additional information to the aspect; it is passed when calling the `ICapi.setAspectContext(Properties props)` method. Some exceptions could be thrown, it depends on the implementer of the aspect if they are used or not.

After you created the new aspect, you add it to the Capi by calling

```
LocalAspect aspect = new LoggingAspect();
List<IPoint> p = new ArrayList<IPoint>();
p.add(LocalIPoint.PostWrite);
p.add(LocalIPoint.PostRead);
capi.addAspect(cref, p, aspect);
```

You pass a list of IPoints to indicate the actions where the aspect offers some special functionality and the aspect itself. The reason that you need to pass a number of IPoints is that you can choose if you want to perform only a part of the actions that an Aspect would offer. Imagine for example a Logging Aspect which offers Logging possibility for each Pre-/Post method, but you only would like to use it for the write action. Instead of writing a new Aspect, you use the IPoints PreWrite and PostWrite and that's it.

As explained in the introduction of Chapter 4, you have the possibility to add multiple Aspects to let them work one after the other. You only need to call the `add` method with the Aspect in the same order as you'd like them to be executed.

4.2 Global Aspects

Using Global Aspects, you can listen to the following actions, additionally to those already mentioned in Local Aspects:

- createContainer()
- destroyContainer()
- transactionCreate()
- transactionCommit()
- transactionRollback()
- coreShutdown()

The usage of a Global Aspect is similar to the usage of a Local Aspect. The only differences are the methods that are offered and that a Global Aspect is associated with a space, not with a specific container. So, for example you overwrite the following method:

```
public void preContainerCreate(
                  String containerName,
                  int size,
                  Properties contextProperties,
                  ICoordinator... coordinators)
              throws AspectNotOkException,
              AspectRescheduleException,
              AspectSkipException
```

and then, you add an Aspect as follows:

```
URI uri = new URI(...); //URI to a space (may be null for local)
GlobalAspect aspect = new SecurityAspect();
List<IPoint> p = new ArrayList<IPoint>();
p.add(GlobalIPoint.PreCoreShutdown);
capi.addAspect(uri, p, aspect);
```

```
public static void main(String[] args)
{

      try
      {
            ICapi capi = new Capi();
            /* Create new Container using a container-size of 3 */
            ContainerRef cref = capi.createContainer(null, null, null, 3,
                        new FifoCoordinator());

            LocalAspect aspect = new LoggingAspect();
            /* Create the IPoints for the aspect */
            List<LocalIPoint> p = new ArrayList<LocalIPoint>();
            p.add(LocalIPoint.PostWrite);
            p.add(LocalIPoint.PostRead);
```

```
            capi.addAspect(cref, p, aspect);

            /* Write 3 entries to the container */
            for (int i = 1; i <= 3; i++)
            {
                  Entry entry = new AtomicEntry<Integer>(i);
                  capi.write(cref, 0, null, entry);
            }
            capi.read(cref, 0, null);

            capi.shutdown(null, true);
      }
      catch (Exception e)
      {
            e.printStackTrace();
      }
}
```
Listing 4.2: Adding the LoggingAspect to Capi

You also have the possibility to use the LocalAspects together with GlobalAspects-. In this case, the actions that are specific for LocalAspects are performed on ALL containers and not only a specific one. Using this functionality, you can for example implement a method that writes a log entry whenever an entry is read from ANY container in your space.

Summary:

Using Aspects, you can easily add new functionality to your XVSM system. You can define some actions that are done either before or after an operation is done with the container or space. So for example you can add the functionality that before a write, the permissions of the writer is checked. This way, security checks can be introduced. For a new Aspect, you overwrite the pre- and post-methods to add new functionality and then, you register the Aspect in the Capi.

Chapter 5:
Don't miss a thing: Notifications

Using Aspects, a very important feature was added to the XVSM implementation: Notifications. With notifications, you can listen on various actions and be called back if such an action is performed. For example:
You have a listening class that would like to be notified if a write is called (and thus, a new entry is added). This class must implement the `NotificationListener` interface and so it must implement the method

```
void handleNotification(final NotificationContext n)
```

Then you need to register this class as listener for the call of the write method:

```
capi.createNotification(
        cref,
        Capi.INFINITE,          //how often the notification should fire
        NotificationTarget.WRITE,
        true,                   //give entries to notified class
        this,                   //the class to be notified
        (Selector[]) null);
```

If you would like to listen to the usage of specific entries, you can use a Selector – it is internally used just as if you would select the entries with the read() or take() methods. In the current version, this behaviour is not yet supported.
The maximum number of times the Notification should fire can be used in such a way to let the class that implements NotificationListener only notify a number of times, until the notification is removed. This could be intended if you're only interested in a small set of entries that you're listening to. Each time the notification is fired, an internal number is increased. If the number is higher than or equal the maximum number, the NotificationListener is removed from the Capi. If you don't want to have the NotificationListener removed, use Capi.INFINITE as parameter.

There exist the following notification targets, which should be self-explanatory:

```
NotificationTarget.WRITE
NotificationTarget.READ
NotificationTarget.TAKE
NotificationTarget.SHIFT
NotificationTarget.DESTROY
```

Putting all together, you can find that with the usage of notifications, you can much more efficiently program some of the examples/exercises of the recent chapters. Just to take one example:

In the TicketQueue example, you've created one main peer which handles the Persons in the queue one after the other. Now imagine that you start one peer for each person and one peer for the shop assistant (which now is a new class). Both the Person and the Shop assistant implement the NotificationListener interface. When the Shop assistant is instantiated, it is registered as listener on a request to pay a ticket. The Person is registered as listener of the ticket when it has paid the ticket to receive the printed ticket.

And as you might expect, there is a final exercise:

5.1 Exercise (TicketQueue with Notifications)

Try to program the just-described TicketQueue Example. You will need to have

- A Ticket and a Payment object that are used to notify the other party to do something.
- A second container to store these Ticket and Payment entries independently from the FIFO queue of the persons. You may name this container "SalesDesk".
- A couple of Persons which have a `payTicket()` method – the `waitForTicket()` method is not needed anymore, as you now have a handleNotification() method which tells the Person that the Ticket is printed. In the `payTicket()` method, write the Payment object to the SalesDesk container and register the Person as listener for a call of the write method. In the handleNotification method, you need to take care if really a Ticket was returned (imagine multiple ShopAssistants and Persons at a time writing to the container – if you don't check if the Ticket is "yours", the person could either "steal" Payment or Tickets from others). If it really is a ticket, take it from the container, give a System.out.println that you've got the ticket and leave the queue ("destroy" the Person object from the container and shutdown this peer).
- A single ShopAssistant class which implements NotificationListener. In its handleNotification method, take the Payment from the SalesDesk container,

give a `System.out.println()` that you're printing the ticket and wait for 1 second. Then store the Ticket object in the container. Again, in the handleNotification method, take care that you only take Payment objects, not Ticket objects.

Summary:

A Notification is used if your class is interested in a special event in the Container, for example if an Entry was written or read. This Entry can be passed to the notify method in order to enable the listening class to perform some action using this Entry.

Using Notifications and looking at the last example, the character of Space-based computing is shown best: Instead of imperatively telling the ShopAssistant or the Person to do something, you write a request to the space instead. The other party takes the request, performs the requested action and writes the result back. If the requested party is overloaded or too slow – no problem: you add one more of such participants; they collaboratively work on the requests in the container. With the possibility to spread the container over multiple peers and with help of transactions, you can implement load balancing scenarios in the easiest way. Moreover, you can add or remove working peers on-the-fly.

APPENDIX: Solutions for the exercises

(only available for those who really tried hard to solve the problem)

The TicketQueue (see Exercise 2.6.2):

```
public class TicketQueue
{
   public static void main(String[] args)
   {
      try
      {
         ICapi capi = new Capi();
         FifoSelector sel = new FifoSelector();

         /*
          * Create new Container using FifoCoordinator, FifoSelector and an
          * infinite container-size
          */
         ContainerRef cref = capi.createContainer(null, null,
            null,
            IContainer.INFINITE_SIZE,
            new FifoCoordinator());

         /* Fill the queue with a couple of persons */
         for (int i = 1; i <= 5; i++)
         {
            Entry entry = new AtomicEntry<Person>(new Person(i));
            capi.write(cref, 0, null, entry);
         }

         try
         {
            /* Handle the persons of the queue */
            while (true)
            {
               /* Take an entry from the container using the FifoSelector */
               Entry[] readEntries = capi.take(cref, 0, null, sel);
               if (readEntries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
               {
                  Person actualPerson = ((AtomicEntry<Person>) readEntries[0]).getValue();
                  /* start paying the Ticket */
                  actualPerson.payTicket();
                  /* start printing the Ticket */
                  actualPerson.waitForTicket();
                  System.out.println(
                     "The " + actualPerson.waitingPosition + ". person received his ticket");
               }
            }
         }
         catch(Exception e)
         { //do nothing - this happens when the container is empty
         }

         capi.shutdown(null,true);

         System.exit(0);
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

TicketQueue.java

```
public class Person
{
   int waitingPosition;

   public Person(int waitingPosition)
   {
      this.waitingPosition = waitingPosition;
   }

   /**
    * Simulate payment
    */
   public void payTicket()
   {
      try
      {
         System.out.println("The " + waitingPosition + ". person pays the ticket.");
         Thread.sleep(1000);
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }

   /**
    * Simulate waiting for he ticket
    */
   public void waitForTicket()
   {
      try
      {
         System.out.println("The " + waitingPosition + ". person waits for the ticket.");
         Thread.sleep(1000);
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

Person.java

Formula 1 Race (see Exercise 2.7.2):

```
public class Formula1Race
{
   final protected static int MAX_LAPS = 2;

   public static void main(String[] args)
   {
      String driver[] = {"Lewis Hamilton", "Fernando Alonso", "Kimi Räikkönen",
            "Felipe Massa", "Nick Heidfeld", "Robert Kubica",      "Heikki Kovalainen",
            "Giancarlo Fisichella", "Nico Rosberg", "Alexander Wurz"};

      try
      {
         ICapi capi = new Capi();

         /* Create new Container using VectorCoordinator and VectoSelector */
         ContainerRef racingPositions = capi.createContainer(null,
            null,
            null,
            IContainer.INFINITE_SIZE,
            new VectorCoordinator());
         ContainerRef retiredCars = capi.createContainer(null,
            null,
            null,
            IContainer.INFINITE_SIZE,
            new VectorCoordinator());
```

```
        /* Create 10 RacingCar instances and write them to the container */
        for (int i = 0; i < driver.length; i++)
        {
           RacingCar writeCar = new RacingCar(driver[i]);
           Entry entry = new AtomicEntry<RacingCar>(writeCar,
              RacingCar.class,
              new VectorSelector(VectorSelector.APPEND, 0));
           capi.write(racingPositions, 0, null, entry);
        }

        printCurrentPositioning(capi, racingPositions, retiredCars, 0);

        /*
         * LAP 1 Car number 6 has an accident and is removed from the list
         */

        /* Read and destroy entry 6 in the container */
        Entry[] readEntries = capi.take(racingPositions, 0, null, new VectorSelector(6, 1));

        /* Write the read entry to the retiredCars container */
        if (readEntries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
        {
           RacingCar readCar = ((AtomicEntry<RacingCar>) readEntries[0]).getValue();
           Entry entry = new AtomicEntry<RacingCar>(readCar,
              RacingCar.class,
              new VectorSelector(0));
           capi.write(retiredCars, 0, null, entry);
        }

        printCurrentPositioning(capi, racingPositions, retiredCars, 1);

        /*
         * LAP 2 Car number 4 overtakes number 3
         */

        /* Read and destroy entry 4 in the container */
        readEntries = capi.take(racingPositions, 0, null, new VectorSelector(4, 1));

        if (readEntries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
        {
           /* Write back previously read entry at index 3 */
           RacingCar readCar = ((AtomicEntry<RacingCar>) readEntries[0]).getValue();
           Entry entry = new AtomicEntry<RacingCar>(readCar,
              RacingCar.class,
              new VectorSelector(3, 0));
           capi.write(racingPositions, 0, null, entry);
        }

        printCurrentPositioning(capi, racingPositions, retiredCars, 2);

        /* Shutdown and clear space */
        capi.shutdown(null,true);
     }
     catch (Exception e)
     {
        e.printStackTrace();
     }

     System.exit(0);
  }

  /*
   * Print current positioning
   */
  public static void printCurrentPositioning(ICapi capi, ContainerRef racingPostions,
           ContainerRef retiredCars, int lap) throws Exception
  {
     if (lap == 0)
        System.out.println("*** Starting Grid ***");
     else if (lap == MAX_LAPS)
        System.out.println("*** Lap " + lap + " - Final Lap ***");
     else
        System.out.println("*** Lap " + lap + " ***");
```

```
      /* Read the current positioning */
      Entry[] readEntries = capi.read(racingPostions, 0, null, new
VectorSelector(VectorSelector.CNT_ALL));

      int i = 1;
      for (Entry readEntry : readEntries)
      {
         if (readEntry.getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
         {
            RacingCar car = ((AtomicEntry<RacingCar>) readEntry).getValue();
            System.out.println("Pos " + i++ + " - " + car.getDriver());
         }
      }

      /* Read all retired cars */
      readEntries = capi.read(retiredCars, 0, null, new VectorSelector(Selector.CNT_ALL));

      for (Entry readEntry : readEntries)
      {
         if (readEntry.getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
         {
            RacingCar retiredCar = ((AtomicEntry<RacingCar>) readEntry).getValue();
            System.out.println("Retired " + retiredCar.getDriver());
         }
      }

      System.out.println("*** ****** ****** ***\n");
   }
}
```

Formula1Race.java

```
public class RacingCar
{
   private String driver = null;

   public RacingCar(String driver) throws Exception
   {
      if ((driver == null) || (driver.length() == 0))
         throw new Exception("Driver must have a length greater 0!");

      this.driver = driver;
   }

   public String getDriver()
   {
      return driver;
   }
}
```

RacingCar.java

Formula1Race extended by Transactions (see Exercise 3.1)

```
public class Formula1RaceTransaction extends Thread
{
   public static final String RACING_POSITIONS_CONTAINER = "RacingPositions";
   public static final String COORDINATION_CONTAINER = "Coordination";
   public static final int MAX_LAPS = 2;

   private ICapi capi = null;
   private ContainerRef racingPositions = null;
   private ContainerRef coordination = null;

   private boolean wait = true;


   public Formula1RaceTransaction()
   {}

   /**
    * Open and manage a race
    *
    */
   public void openRace()
   {
      try
      {
         capi = new Capi();

         racingPositions = capi.createContainer(null,
               null,
               RACING_POSITIONS_CONTAINER,
               10,
               new VectorCoordinator());
         coordination = capi.createContainer(null,
               null,
               COORDINATION_CONTAINER,
               IContainer.INFINITE_SIZE,
               new VectorCoordinator());

         /* Start waiting thread
          *
          * Note:
          * This is necessary because there is a bug in the XVSM-CORE-API, if
          * you set the TimeOut to 10000 (= 10sec.) the next read/write on the
          * same container blocks.
          *
          * Correct code would be:
          * capi.read(racingPositions, 10000, null, new VectorSelector(10));
          *
          * Which means: Either wait until 10 objects are in the container or
          * the 10seconds are over. In the case of a timeout an exception will
          * be thrown. This is no problem because we read the positions again
          * counting all objects in the container.
          */
         start();

         int registeredCars = 0;

         while(wait)
         {
            try
            {
               /* Read positions, are already 10 objects available? */
               capi.read(racingPositions, 0, null, new VectorSelector(10));
               wait = false;
               System.out.println("Okay! 10 cars!");
            }
            catch (Exception e)
            {
               /* A "CountNotMetException is thrown, but we are not interested in */
            }
         }
```

```
         /* read and count how many objects are in the container */
         registeredCars = capi.read(racingPositions,
               0,
               null,
               new VectorSelector(Selector.CNT_ALL)).length;

         System.out.println("Starting Race! " + registeredCars);

         /* Write "start-race"-message to the container */
         Entry startMessage = new AtomicEntry<Message>(
               new Message(Message.START_RACE, null),
               Message.class,
               new VectorSelector(VectorSelector.APPEND, 0));
         capi.write(coordination, Capi.INFINITE_TIMEOUT, null, startMessage);

         /* Wait until all cars have finished */
         capi.read(coordination,
               Capi.INFINITE_TIMEOUT,
               null,
               new VectorSelector(1 + registeredCars));

         System.out.println("Race finished!\n Shuting down now!");

         capi.shutdown(null, true);
         System.exit(0);
      }
      catch (Exception e)
      {
         e.printStackTrace();
         System.out.println("An error has occured, shutting-down!");
         try
         {
            /* Try to shutdown Capi */
            capi.shutdown(null, true);
         }
         catch (XCoreException e1)
         {
            e1.printStackTrace();
         }
         System.exit(0);
      }
   }

   /**
    * A thread waiting 15 seconds
    *
    */
   public void run()
   {
      for (int i = 15; i > 0; i--)
      {
         System.out.println("Waiting " + i + " seconds for Clients");
         try
         {
            Thread.sleep(1000);
         }
         catch (Exception e)
         {
            e.printStackTrace();
         }
      }

      wait = false;
   }

   public static void main(String[] args)
   {
      Formula1RaceTransaction f1Race = new Formula1RaceTransaction();
      f1Race.openRace();
   }
}
```

Formula1RaceTransaction.java

```
public class RacingCar
{
   /**
    * RacingCar requires one parameter at start, the name of
    * it's "driver". It is up to you to provide unique names
    * when running several clients on one server.
    */
   public static void main(String[] args)
   {
      if ((args.length != 1))
      {
         System.out.println("Usage: RacingCar name\n" +
                            "name ... Driver-Name");
         System.exit(-1);
      }

      String driver = args[0];

      try
      {
         ICapi capi = new Capi();

         ContainerRef racingPositions = capi.lookupContainer(
               null,
               new URI("tcpjava://localhost:1234"),
               Formula1RaceTransaction.RACING_POSITIONS_CONTAINER);
         ContainerRef coordination = capi.lookupContainer(
               null,
               new URI("tcpjava://localhost:1234"),
               Formula1RaceTransaction.COORDINATION_CONTAINER);
         System.out.println("* Found containers");

         /* Write our CarInfo into the positions container */
         CarInfo currentCar = new CarInfo(driver);
         Entry carInfo = new AtomicEntry<CarInfo>(currentCar,
               CarInfo.class,
               new VectorSelector(VectorSelector.APPEND, 0));
         capi.write(racingPositions, 0, null, carInfo);
         System.out.println("* Wrote CarInfo");

         System.out.println("* Waiting");

         /* Wait until we receive the "race-start"-message
          *
          * NOTE:
          * Normaly it would be enough to wait until the first
          * element is written to the container, but we discovered
          * that there is still a bug in the XVSM-CORE-API when several
          * concurrent reads are performed (this happens when more than
          * one client is trying to read).
          * The correct code would be:
          * capi.read(coordination, Capi.INFINITE_TIMEOUT, null, new VectorSelector(1));
          * instead of "for-loop"
          */
         for(;;)
         {
            try
            {
               capi.read(coordination, 0, null, new VectorSelector(1));
               break;
            }
            catch (Exception e)
            {
            }
         }

         System.out.println("* Race Started");

         /* Simulate Formula1RaceTransaction.MAX_LAPS laps */
         for (int lap = 0; lap < Formula1RaceTransaction.MAX_LAPS; lap++)
         {
            /* wait for a random-time */
            Random rand = new Random();
            int time = rand.nextInt(6) + 3;
```

```
            try
            {
               System.out.println("Simulating lap, length: " + time + "sec.");
               Thread.sleep(time * 1000);
            }
            catch (InterruptedException e)
            {
               e.printStackTrace();
            }

            boolean commited = false;
            /* We need to retry our Transaction until we are able to commit it */
            while (!commited)
            {
               try
               {
                  /* Create a transaction */
                  Transaction tx = capi.createTransaction(
                      new URI("tcpjava://localhost:1234"),
                      Capi.INFINITE_TIMEOUT);
                  System.out.println("* Transaction created");

                  /* Read the actual positions using transaction */
                  Entry[] positions = capi.read(
                      racingPositions,
                      0,
                      tx,
                      new VectorSelector(Selector.CNT_ALL));

                  int pos = 0;
                  CarInfo car = null;

                  /* Search for the own CarInfo */
                  for (Entry position : positions)
                  {
                     if (position.getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
                     {
                      car = ((AtomicEntry<CarInfo>) position).getValue();
                      if (car.getDriver().compareTo(driver) == 0)
                      {
                              /* Remove the CarInfo from the list using transaction */
                              capi.take(racingPositions, 0, tx, new VectorSelector(pos, 1));
                              break;
                      }
                     }

                     pos++;
                  }
                  currentCar = car;
                  currentCar.addTime(time);
                  currentCar.incrementLap();

                  System.out.println("* Updated my CarInfo");

                  /* Read the actual positions again using transaction */
                  positions = capi.read(
                      racingPositions,
                      0,
                      tx,
                      new VectorSelector(Selector.CNT_ALL));

                  /* Find the right place where to position the own CarInfo */
                  if (positions.length > 0)
                  {
                     for (pos = 0; pos < (positions.length); pos++)
                     {
                      if (positions[pos].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
                      {
                              car = ((AtomicEntry<CarInfo>) positions[pos]).getValue();
                              if (car.getTotalTime() > currentCar.getTotalTime())
                              {
                                      /* Create a new entry */
                                      Entry entry = new AtomicEntry<CarInfo>(
                                                      currentCar,
                                                      CarInfo.class,
                                                      new VectorSelector(pos, 1));
```

```
                                 /* Write entry to the container using transaction */
                                 capi.write(racingPositions, 0, tx, entry);
                                 break;
                              }
                              /* we reached the end of the positions */
                           else if (pos == (positions.length - 1))
                           {
                              /* Create a new entry */
                              Entry entry = new AtomicEntry<CarInfo>(
                                             currentCar,
                                             CarInfo.class,
                                             new VectorSelector(VectorSelector.APPEND, 0));
                              /* Write entry to the container using transaction */
                              capi.write(racingPositions, 0, tx, entry);
                              break;
                           }
                      }
                      pos++;
                     }
                  }
                  else /* position-list is empty, we have to append the CarInfo */
                  {
                     /* Create a new entry */
                     Entry entry = new AtomicEntry<CarInfo>(
                              currentCar,
                              CarInfo.class,
                              new VectorSelector(VectorSelector.APPEND, 0));
                     /* Write entry to the container using transaction */
                     capi.write(racingPositions, 0, tx, entry);
                  }

                  /* Try to commit the transaction */
                  capi.commitTransaction(tx);
                  commited = true;
                  System.out.println("* Updated positions");
                  System.out.println("* Transaction commited");
               }
               catch (Exception e)
               {
                  /* If concurrent transactions exist on the same container, all
                   * except the first will throw an Exception. We don't have to
                   * worry about it, just have to try our transaction again.
                   *
                   * Uncomment the next line if you want to see it.
                   */
                  //e.printStackTrace();
                  System.out.println("Error in Transaction, retrying...");
                  Thread.sleep(250); // wait before retrying
               }
            }
         }

         /* Read the actual positions using transaction */
         Entry[] positions = capi.read(
               racingPositions,
               Capi.INFINITE_TIMEOUT,
               null,
               new VectorSelector(Selector.CNT_ALL));

         int pos = 1;
         CarInfo car = null;

         /* Print actual position-list*/
         System.out.println("\n*** Positionings after "+Formula1RaceTransaction.MAX_LAPS +
               " laps ***");
         for (Entry position : positions)
         {
            if (position.getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
            {
               car = ((AtomicEntry<CarInfo>) position).getValue();
               System.out.println(pos+". " + car.getDriver() + ", Time: " +
car.getTotalTime());
            }

            pos++;
         }
```

```java
         System.out.println("*** *** *** *** *** *** *** ***\n");

         /* Write message that this RacingCar has finished */
         Entry finishedMsg = new AtomicEntry<Message>(
               new Message(Message.CAR_FINISHED, currentCar),
               Message.class,
               new VectorSelector(VectorSelector.APPEND, 0));
         capi.write(coordination, 0, null, finishedMsg);

         System.out.println("* Finished");

         /* Shutdown Capi */
         capi.shutdown(null, true);
         System.exit(0);
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

RacingCar.java

```java
public class Message implements Serializable
{
   public static final int START_RACE = 0;
   public static final int CAR_FINISHED = 1;

   private int messageType = -1;
   private CarInfo car = null;

   /**
    * "car" can be null, which means that the
    * server has sent a message to the clients
    */
   public Message(int messageType, CarInfo car)
   {
      this.messageType = messageType;
      this.car = car;
   }

   /**
    * Get the message's type
    */
   public int getMessageType()
   {
      return messageType;
   }

   /**
    * Get the stored CarInfo
    */
   public CarInfo getCarInfo()
   {
      return car;
   }
}
```

Message.java

```java
public class CarInfo implements Serializable
{
   private String driver = null;
   private int totalTime = 0;
   private int currentLap = 0;

   /**
    * We need the driver's name to be able
    * to identify it.
    */
   public CarInfo(String driver)
   {
      this.driver = driver;
   }
```

```
    /**
     * Get the driver's name
     */
    public String getDriver()
    {
       return driver;
    }

    /**
     * Get the TotalTime
     */
    public int getTotalTime()
    {
       return totalTime;
    }

    /**
     * Get the lap-count
     */
    public int getCurrentLap()
    {
       return currentLap;
    }

    /**
     * Add time to the totaltime
     */
    public void addTime(int time)
    {
       totalTime += time;
    }

    /**
     * Increase lap-count by 1
     */
    public void incrementLap()
    {
       currentLap++;
    }
}
```

CarInfo.java

TicketQueue extended by Notifications (see Exercise 5.1)

```
public class TicketQueueNotification
{
   public static void main(String[] args)
   {
      boolean wait = true;

      try
      {
         ICapi capi = new Capi();
         ContainerRef salesDesk = capi.createContainer(null, null,
               ISalesDeskObject.SALES_DESK_CONTAINER,
               IContainer.INFINITE_SIZE,
               new FifoCoordinator());

         System.out.println("+ Server started");
         System.out.println("+ To stop server type \"q\" and enter");

         BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
         /* wait until someone wants to shutdown the server */
         while(wait)
         {
            if (stdin.readLine().compareTo("q") == 0)
            {
               wait = false;
            }
         }

         System.out.println("+ Sending stop to all clients");
         /* Write a "ShopClosed"-object to the container */
         Entry entry = new AtomicEntry<ISalesDeskObject>(
               new ShopClosed(ISalesDeskObject.SHOP_OWNER_ID));
         capi.write(salesDesk, capi.INFINITE_TIMEOUT, null, entry);

         /* Give the clients 5 seconds to shutdown */
         for (int i = 5; i > 0; i--)
         {
            System.out.println("+ Shutdown in " + i + " seconds");
            Thread.sleep(1000);
         }

         capi.shutdown(null, true);
         System.exit(0);
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

TicketQueueNotification.java

```
public class ShopAssistant implements NotificationListener
{
   ICapi capi = null;
   boolean shopOpen = true;

   public ShopAssistant()
   {  }

   /**
    * This is where the shop-assistant instantiates the capi,
    * looks-up the salesdesk-container and after registering
    * as a listener it waits until the "Shop", the
    * TicketQueueNotification, sends a "ShopClosed"-object.
    */
   public void run()
   {
      try
      {
         capi = new Capi();
```

```
        /* Look-up SalesDesk-container*/
        ContainerRef salesDesk = capi.lookupContainer(null,
              new URI("tcpjava://localhost:1234"),
              ISalesDeskObject.SALES_DESK_CONTAINER);

        /* Register as a listener to the container*/
        capi.createNotification(salesDesk,
              Capi.INFINITE,
              NotificationTarget.WRITE,
              true, this);
        System.out.println("Found and registered listener to container!");

        /* wait until the shop is closing*/
        while(shopOpen)
        {}

        capi.shutdown(null, true);
     }
     catch (Exception e)
     {
        e.printStackTrace();
     }
  }

  /**
   * Handle a received notification. We are only interested in "Payments" and
   * "ShopClosed" entries. If a payment is received a ticket is placed instead
   * to the container.
   */
  public void handleNotification(NotificationContext notificationContext)
  {
     /* Get the container for which we received the notification */
     ContainerRef salesDesk = notificationContext.getCref();

     try
     {
        /* Read the oldest entry */
        Entry[] entries = capi.read(salesDesk,
              Capi.INFINITE_TIMEOUT,
              null,
              new FifoSelector());

        if (entries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
        {
           ISalesDeskObject sdo = ((AtomicEntry<ISalesDeskObject>) entries[0]).getValue();

           /* Check if the object is a payment */
           if (sdo instanceof Payment)
           {
              /* Take the payment */
              Entry[] payments = capi.take(salesDesk,
                    Capi.INFINITE_TIMEOUT,
                    null,
                    new FifoSelector());
              Payment payment = (Payment) ((AtomicEntry<ISalesDeskObject>)
                    payments[0]).getValue();
              System.out.println(
                 "ShopAssistant: Received Payment from Customer " + payment.getOwner());

              System.out.println(
                 "ShopAssistant: Printing Ticket for Customer " + payment.getOwner());
              Thread.sleep(1000); /* wait 1 second */

              /*Create a ticket for the customer and write it into the container */
              Ticket ticket = new Ticket(payment.getOwner());
              Entry entry = new AtomicEntry<ISalesDeskObject>(ticket);
              capi.write(salesDesk, Capi.INFINITE_TIMEOUT, null, entry);
              System.out.println(
                 "ShopAssistant: Placed Ticket for Customer " + ticket.getOwner());
           }
           /* Check if the object is a "ShopClosed" */
           else if (sdo instanceof ShopClosed)
           {
              System.out.println("ShopAssistant: Received closing shop, exiting now!");
              shopOpen = false;
           }
```

```java
         }
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }

   public static void main(String[] args)
   {
      ShopAssistant assistant = new ShopAssistant();
      assistant.run();

      System.out.println("Exiting!");
      System.exit(0);
   }
}}
```

ShopAssistent.java

```java
public class Customer implements NotificationListener
{
   private int ID = -1;
   private ICapi capi = null;
   private boolean stop = false;

   /**
    * To be able to identify our payment and ticket an unique
    * ID is needed.
    */
   public Customer(int ID)
   {
      this.ID = ID;
   }

   /**
    * This is where the customer instantiates the capi, looks-up
    * the salesdesk-container, places the payment and after
    * registering as a listener it waits until the ticket has been
    * received
    */
   public void buyTicket()
   {
      try
      {
         capi = new Capi();

         /* Look-up SalesDesk-container*/
         ContainerRef salesDesk = capi.lookupContainer(null,
               new URI("tcpjava://localhost:1234"),
               ISalesDeskObject.SALES_DESK_CONTAINER);
         System.out.println("Customer: Found Container");

         /* Write payment into the container*/
         Entry entry = new AtomicEntry<ISalesDeskObject>(new Payment(ID));
         capi.write(salesDesk, Capi.INFINITE_TIMEOUT, null, entry);
         System.out.println("Customer: Placed Payment");

         /* Register as a listener to the container*/
         capi.createNotification(salesDesk,
               Capi.INFINITE,
               NotificationTarget.WRITE,
               true,
               this);
         System.out.println("Customer: Registered listener to container!");

         /* Wait till either we have received our ticket our the shop has closed */
         while(!stop)
         {}

         capi.shutdown(null, true);
      }
```

```
    catch (Exception e)
    {
       e.printStackTrace();
    }
 }

 /**
  * Handle a received notification. We are only interested in "Ticket" and
  * "ShopClosed" entries. If it is a ticket than check if it is our ticket
  * before taking it from the container.
  *
  */
 public void handleNotification(NotificationContext notificationContext)
 {
    /* Get the container for which we received the notification */
    ContainerRef salesDesk = notificationContext.getCref();

    try
    {
       /* Read the oldest entry */
       Entry[] entries = capi.read(salesDesk,
             Capi.INFINITE_TIMEOUT,
             null,
             new FifoSelector());

       if (entries[0].getEntryType().equals(Entry.EntryTypes.ATOMICENTRY))
       {
          ISalesDeskObject sdo = ((AtomicEntry<ISalesDeskObject>) entries[0]).getValue();

          /*Check if the object is a ticket and if it is the one we were so long waiting for*/
          if ((sdo instanceof Ticket) && (((Ticket) sdo).getOwner() == ID))
          {
             /* take our ticket from the Container */
             Entry[] tickets = capi.take(salesDesk,
                   Capi.INFINITE_TIMEOUT,
                   null,
                   new FifoSelector());
             Ticket ticket = (Ticket) ((AtomicEntry<ISalesDeskObject>) tickets[0]).getValue();
             System.out.println("Customer: Received my Ticket!");
             stop = true;
          }
          /* Check if the object is a "ShopClosed" */
          else if (sdo instanceof ShopClosed)
          {
             System.out.println("Customer: Received closing shop, exiting now!");
             stop = true;
          }
       }
    }
    catch (Exception e)
    {
       e.printStackTrace();
    }
 }

 /**
  * main-method, needs 1 integer parameter representing the customer-ID
  * @param args[0] ... customer-ID
  */
 public static void main(String[] args)
 {
    try
    {
       if ((args.length != 1) || (Integer.parseInt(args[0]) <= 0))
       {
          System.out.println("Usage: customer ID\n" +
                   "ID ... unique customer-ID > 0");
          System.exit(-1);
       }
    }
    catch (NumberFormatException e)
    {
       System.out.println("Usage: customer ID\n" +
          "ID ... unique customer-ID > 0");
       System.exit(-1);
    }
```

```java
      Customer customer = new Customer(Integer.parseInt(args[0]));
      customer.buyTicket();

      System.out.println("Exiting!");
      System.exit(0);
   }
}
```

Customer.java

```java
public interface ISalesDeskObject extends Serializable
{
   /* Container-Name needed by the TicketQueueNotification, ShopAssistant and Customer */
   public static final String SALES_DESK_CONTAINER = "SalesDesk";
   public static final int SHOP_OWNER_ID = -1; //

   public int getOwner();
}
```

ISalesObject.java

```java
public class Payment implements ISalesDeskObject
{
   private int owner = 0; // Customer who placed this payment

   /**
    * Create a payment
    */
   public Payment(int owner)
   {
      super();
      this.owner = owner;
   }

   /**
    * return this payment's owner
    */
   public int getOwner()
   {
      return owner;
   }
}
```

Payment.java

```java
public class ShopClosed implements ISalesDeskObject
{
   private int owner; // in this case the shop (server)

   /**
    * Create a ShopClosed
    */
   public ShopClosed(int owner)
   {
      this.owner = owner;
   }

   /**
    * return the owner
    */
   public int getOwner()
   {
      return owner;
   }
}
```

ShopClosed.java

```
public class Ticket implements ISalesDeskObject
{
   int owner = 0; // Customer who payed for this ticket

   /**
    * Create a payment
    * @param person ... The person who has payed for the Ticket
    */
   public Ticket(int owner)
   {
      this.owner = owner;
   }

   /**
    * return this payment's owner
    */
   public int getOwner()
   {
      return owner;
   }
}
```
Ticket.java

Summary

XVSM (eXtensible Virtual Shared Memory), developed at the Institute of Computer Languages at the Technical University of Vienna, is middleware that offers the possibility to store data in a common data space. This data space can be used as shared memory of the participants that are connected to this space. They have the possibility to write, read, take and delete these space entries. Also, it is possible to register on actions that are performed on the space to react on such situations. For example, it then is possible to call a certain function when a data item is added to the space. This feature can help to enhance the functionality of the system even while runtime, as such listeners can be registered while the system is running. It is planned for future versions of XVSM to support the distribution of data over multiple peers and to persist the data.

The framework is still in development, some tasks could already be accomplished, others can't be done in the current version. In this master thesis, I would like to give newbies an introduction to the XVSM programming interface, whereas I would like to focus on tasks that could already be done using the MozartSpaces version 1.0, which is an implementation of the XVSM programming interface in Java 5.0. This means that some tasks that are prepared in XVSM, like the distribution of data, that I just mentioned, are not functioning at the moment, whereas others are implemented already. I will focus on the functions that are already implemented and can be used by the reader of my tutorial right away.

My master thesis consists of three parts:

- the Master Thesis Main document, which contains meta-information of the tutorial and the application scenarios; it is the document you are currently reading. It is mainly focused on the XVSM and MozartSpaces developers to keep the Tutorial up-to-date.
- the Application Scenarios which is a document aimed to designers and/or developers who want to know how to make various architectural decisions using XVSM. The document describes the change in the paradigm from client-server architecture to space-based computing. It moreover shows some

application possibilities of XVSM, namely the possibility to cooperatively work on the data items to improve performance, to handle high-priority data items and to construct chains of calculations, also known as SEDA approach [6]. These chains can be used to construct automatic load balancing of incoming requests and automatic recovery after failure.

- the Tutorial, which is the document that can be given to software developers who are interested in learning XVSM, not in reading my complete diploma thesis. This is the technical part, where the reader learns how to program using XVSM. It consists of explanations of the objects, methods and their functionality and behaviour by describing them and by giving examples. Moreover, to achieve better learning results and to challenge the reader to try the explained section, exercises are given that partially are based on preceding exercises or examples. When having finished to work with this document, the reader should be able to use the core functionality to manage the space and the contained objects, to register listeners that are notified if a special action occurs and to listen on new data items in the space. At the end of this document, sample solutions of the exercises as well as the complete example source code are listed.

By the combination of these three documents, I can achieve the needs of three groups of people:

- The developers of the MozartSpaces/XVSM need to find a way to easily keep the Tutorial up-to-date. Therefore, I need to have this Main document, where I give possibilities to quickly find references to the implementation. When the implementation changes, they just need to consult the Main document.
- The program designer, who would like to understand the possibilities of Space Based Computing and XVSM in particular.
- And finally the programmer who is not included in the design process, but who wants to know how to program with XVSM.

One additional assignment of task was to comprehensively explain the Space Based Computing paradigm and to create an introduction to the new XVSM system that can be used for teaching purposes. This results to the tutorial document as mentioned above. This tutorial was practically proved by handing it out to approximately 80

students and the feedback was continuously incorporated into the tutorial to check and improve its usability.

This feedback phase showed that the use of the tutorial is sufficient to understand how program using XVSM. Thus, this task was accomplished.

Used Literature and References

[1] A Survey of Peer-to-Peer Content Distribution Technologies
S. Androutsellis-Theotokis, D. Spinellis
Athens University of Economics and Business
ACM Computing Surveys, Vol. 36, No. 4, December 2004, pp. 335–371.

[2] Gnutella, in:
Wikipedia
http://en.wikipedia.org/wiki/Gnutella
last visited: 18.02.2008

[3] Getting Started With JavaSpaces Technology:
Beyond Conventional Distributed Programming Paradigms
Qusay H. Mamoud, Sun Microsystems Inc.
http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/
12.06.2005

[4] GigaSpaces XAP – Key Capabilities and Components
http://www.gigaspaces.com/pr_key.html
last visited: 25.02.2008

[5] T Spaces
P. Wyckoff
http://www.research.ibm.com/journal/sj/373/wyckoff.html
06.04.1998

[6] SEDA: An Architecture for Highly Concurrent Server Applications
Matt Welsh, Harvard University
http://www.eecs.harvard.edu/~mdw/papers/quals-seda.pdf
09.05.2006

[7] XMLSpaces for Coordination in Web-based Systems
Tolksdorf, Glaubitz
http://www.ag-nbi.de/research/xmlspaces/XMLspacesWETICE2001.pdf
22.02.2001

[9] JavaSpaces, Ähnliche Konzepte (Similar Concepts)
http://de.wikipedia.or/wiki/JavaSpaces#.C3.84hnliche_Konzepte
last visited: 28.07.2007

[10] Sun Microsystems Inc., Java 6.0
http://java.sun.com/javase/6/
last visited: 12.10.2007

[11] Chapter 27 from Ralf Westphal in
Gernot Starke, Stefan Tilkov
SOA-Expertenwissen
dpunkt.verlag
http://www.soa-expertenwissen.de/files/SOA-Expertenwissen_Kapitel_27.pdf

[12] Post-Client/Server Coordination Tools From eva Kühn, Georg Nozicka, In:
Coordination Technology for Collaborative Applications,
Wolfram Cohen, Gustaf Neumann (eds.),
Springer Series Lecture Notes in Computer Science, 1998.

[13] Virtual Shared Memory for Distributed Architecture,
eva Kühn,
Nova Science Publishers, 2001.

[14] Design und Implementierung einer grafischen Komponente für Monitoring von SEDA-Applikationen,
Michael Lafite
Institute of Computer Languages, Techn. University Vienna,
January 2008.

[15] Invalid Message Channel, In:
Enterprise Integration Patterns
Gregor Hohpe, Bobby Woolf,
Addison-Wesley, 2003, pg 117

[16] Integration of XVSM Spaces with the Web to Meet the Challenging Interaction Demands in Pervasive Scenarios,
eva Kühn, Johannes Riemer, Richard Mordinyi, Lukas Lechner
Ubiquitous Computing And Communication Journal (UbiCC), special issue on "Coordination in Pervasive Environments", Vol. 3, ISSN 1992-8424, March, 2008

[17] http://www.complang.tuwien.ac.at/eva/SBC-Group/sbcGroupIndex.html#SBC-Group-XVSM, last visited: February 2008

[18] Linda (coordination language)
Wikipedia
http://en.wikipedia.org/wiki/Linda_%28coordination_language%29
last visited: 11.03.2008

[19] LIME in a Nutshell
Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman
http://lime.sourceforge.net/Lime/nutshell.html
last visited: 30.03.2008

All pictures were drawn by myself, partially with the help of Microsoft Visio 2003, including its contained images / cliparts.

Erklärung

Name: Michael Wittmann

Matr.-Nr.: 0026159

Ich versichere wahrheitsgemäß, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die von mir angegebenen Quellen, Hilfsmittel und Programmteile von Studierenden benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht.

Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Wien, am .... Unterschrift:...........................

Ich erkläre mich mit einer späteren Veröffentlichung meiner Masterarbeit sowohl auszugsweise, als auch als Gesamtwerk in der Institutsreihe oder zu Darstellungszwecken im Rahmen der Öffentlichkeitsarbeit des Institutes einverstanden.

Unterschrift:...........................