



FAKULTÄT FÜR **INFORMATIK**

Design and Implementation of TinySpaces

The .NET Micro Framework based Implementation of XVSM for Embedded Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Alexander Marek

Matrikelnummer 0726166

an der

Fakultät für Informatik der Technischen Universität Wien

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

Wien, 08.02.2010

(Unterschrift Verfasser/in)

(Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

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

Ort, Datum, Unterschrift

Abstract

Until today developing software for embedded devices has been a tedious task with the main problem that those applications interface directly with the hardware they are run on. This causes a strong coupling between hardware and software, thus making it hard to impossible to reuse code.

Furthermore there is a continuously growing number of networked embedded devices which need to collaborate with each other using different communication protocols like TCP/IP, ZigBee, Bluetooth et cetera.

For that reason the need for a common middleware to connect those devices increases, but the tight software-hardware coupling makes it hard to write such a system for different devices. There already exist some middlewares like emORB [1], which is based on CORBA (Common Object Request Broker Architecture). However CORBA does not allow for P2P (Peer-to-Peer) communication and is thus limited for the usage in mobile networked embedded systems.

Back in 2001 Microsoft started the Smart Personal Object Technology (SPOT) initiative and the .NET Micro Framework was born, which made it possible to write managed, hardware-independent code for embedded devices. Even though this framework is only supported on 32 bit devices, it supports a wide range of those and allows for developing a platform independent prototype of a slimmed XVSM middleware.

This thesis focuses on the implementation of TinySpaces, a middleware based on the XVSM (**eX**tensible **V**irtual **S**hared **M**emory) principle. As TinySpaces is specialized for resource constrained devices several compromises need to be made, which are explained in this document. Nevertheless it is shown that TinySpaces is a compatible subset of other XVSM implementations, as it complies with the XVSM standard although several functionalities needed to be slimmed or omitted to make TinySpaces lightweight enough for embedded devices.

To prove that TinySpaces perform well, benchmarks are made concerning memory utilization and CPU usage of TinySpaces, as well as code-size, performance, and byte usage of three implemented serialization mechanisms.

Kurzfassung

Bis heute stellt das Entwickeln von Software für Embedded Devices eine Herausforderung dar. Besonders problematisch ist, dass jene Applikationen direkt auf die Hardware des Gerätes, auf dem sie ausgeführt werden, zugreifen. Dadurch entsteht eine starke Bindung zwischen Software und Hardware, durch welche es schwer bis unmöglich ist, Code wiederzuverwenden.

Zusätzlich steigt die Anzahl an Embedded Devices, welche über ein Netzwerk mittels verschiedenster Kommunikationsprotokolle wie TCP/IP, ZigBee, Bluetooth, etc. miteinander kollaborieren müssen, kontinuierlich.

Aus diesem Grund wird der Bedarf einer einheitlichen Middleware, mit welcher diese Geräte verbunden werden können, immer stärker, doch die starke Bindung zwischen Hardware und Software macht dieses Unterfangen sehr aufwendig. Es existieren bereits Middlewares für Embedded Systems. Diese basieren größtenteils auf CORBA wie auch beispielsweise emORB [1]. Allerdings unterstützt CORBA keine P2P Kommunikation und ist aufgrund dessen nur bedingt geeignet für mobile Networked Embedded Devices.

Im Jahr 2001 startete Microsoft die Smart Personal Object Technology (SPOT) Initiative und das .NET Micro Framework wurde geboren, welches es möglich machte gemanagten und Hardware-unabhängigen Code für Embedded Devices zu verwenden. Dieses Framework kann allerdings lediglich auf 32 bit Geräten ausgeführt werden, dafür wird bis dato bereits eine Vielzahl dieser unterstützt. Aus diesem Grund ist dieses Framework gut geeignet für die Entwicklung eines Plattform-unabhängigen Prototyps einer reduzierten XVSM Middleware für Embedded Devices.

Der Schwerpunkt dieser Diplomarbeit liegt auf der Implementierung von TinySpaces, einer Middleware, die auf dem XVSM (eXtensible Virtual Shared Memory) Prinzip basiert. Da diese auf Geräte mit eingeschränkten Ressourcen spezialisiert ist, müssen mehrere Kompromisse eingegangen werden, die in dieser Arbeit erläutert werden. Es kann gezeigt werden, dass TinySpaces mit einer Untermenge der vollen XVSM Spezifikation kompatibel ist, da einige Funktionalitäten reduziert und/oder ausgespart werden mussten, damit diese Middleware leichtgewichtig genug für Embedded Devices bleibt.

Um zu zeigen, dass TinySpaces auf Embedded Devices performant läuft, werden Tests in Hinblick auf CPU- und Arbeitsspeicherauslastung von TinySpaces, sowie die Code-Size, Performanz und den Byte Verbrauch von drei implementierten Serialisierungsmechanismen, durchgeführt.

Acknowledgements

The accomplishment of this had not been possible without the support of several persons.

First of all I want to thank my supervisor eva Kühn for relying on me. Although she had been involved in numerous research projects and had multiple graduands to conduct, she always found time for giving me feedback and support.

I also want to thank the members of the XVSM Technical Board, which is a periodical meeting on every Thursday forenoon, for their insights into XVSM, their criticism and discussion on contributed ideas, for delivering ideas and the motivation that arouse in me and remained because of that.

Next special thanks goes to Stefan Craß, a colleague and also member of the technical board, as he worked hard on a formal specification on XVSM and delivered great insight to me, and also for the many discussions we had which always led to useful ideas on both sides.

I also want to thank my parents for giving me the chance to study and supporting me whenever time, money or motivation went short and my sister for being such a great model for me.

Last but not least I want to thank my beloved girlfriend for her patience in those many evenings she had to spend alone, as I was working on this thesis.

Contents

Erklärung zur Verfassung der Arbeit	2
Abstract.....	3
Kurzfassung.....	4
Acknowledgements	5
Contents	6
1 Introduction.....	9
2 Objectives and Overview.....	11
3 Space Based Computing Middleware	12
1.1 Excursion: Remote Procedure Calls (RPC)	12
1.2 Excursion: Message passing.....	12
1.3 Excursion: Message Queuing.....	13
1.4 JavaSpaces	14
1.5 XVSM.....	15
1.6 Layered Architecture of XVSM	17
1.6.1 Algebraic Data Structures.....	17
1.6.2 CAPI-1: Basic Operations.....	17
1.6.3 CAPI-2: Transactions.....	18
1.6.4 CAPI-3: Coordination.....	19
1.6.5 CAPI-4: Aspects	21
1.6.6 CAPI-5: XVSM Runtime (with Timeouts).....	22
1.6.7 XVSM Protocol	24
1.6.8 Language Binding (API)	24
2 Embedded Development and the .NET Micro Framework	24
2.1 Embedded Systems	24
2.1.1 Application Areas of Networked Embedded Systems	26
2.2 The .NET Micro Framework	27
2.2.1 The Layered Architecture of the .NET Micro Framework.....	28
2.2.2 Alternatives to the .NET Micro Framework.....	29
2.2.3 Other Microsoft embedded platforms	30

3	Related Work.....	31
4	TinySpaces Design	34
4.1	Contract First Design in TinySpaces	34
4.2	CAPI-1: Basic Operations.....	35
4.2.1	Contracts	35
4.2.2	Implementation	39
4.3	CAPI-2: Transactions	42
4.3.1	Contracts	42
4.3.2	Implementation	45
4.4	CAPI-3: Coordination.....	53
4.4.1	Contracts	54
4.4.2	Implementation	59
4.5	CAPI-4: Aspects	64
4.5.1	Contracts	64
4.5.2	Implementation	69
4.5.3	Notifications	70
4.6	CAPI-5: Runtime.....	75
4.6.1	Contracts	75
4.6.2	Implementation	75
4.6.3	TimeoutHandler	82
4.7	CAPI-5b: Communication	83
4.7.1	Contracts	85
4.7.2	Implementation	87
5	Benchmark	88
5.1	Performance Benchmark.....	88
5.2	Memory Utilization	91
5.3	Power Consumption	92
5.4	Serialization Performance	92
6	Future Work and Ideas.....	96
7	Conclusion.....	98

Abbreviations	99
Table of Figures.....	101
References	104

1 Introduction

Until today, developing software for embedded systems is a challenging task. Embedded developers are used to programming them with C++, C or even assembly language and in addition, the code always interfaces with the hardware directly. As every board has different interrupt controllers, buses and I/O interfaces, etc., developing software which can be ported to any other device is hard to impossible. [2]

Additionally the biggest factor influencing the choice of a processor today is neither its speed, nor its price, but the available software, which eases the development of software for embedded systems. [3]

Another problem is that these devices often need to collaborate with other ones in some networked environment. There already exist several middlewares for embedded devices, but as they are all based on CORBA, their support for P2P communication is very limited. However, this style of communication is important for scenarios where such devices are mobile, like cooperating robotic vacuum cleaners, just to give an example.

This leads to the need of a middleware capable of P2P for interconnecting those devices, but as software is not independent from the underlying hardware, this is a tedious task.

Back in 2001 Microsoft started the Smart Personal Object Technology (SPOT) initiative, “aimed at improving the function of everyday objects through the injection of software.” [4]

“Smart Personal Objects are everyday objects, such as clocks, pens, key-chains and billfolds that are made smarter, more personalized and more useful through the use of special software.” [4]

The first outcome of this initiative was an ECMA-compliant (European Computer Manufacturers Association) subset of the CLR (Common Language Runtime) which is called TinyCLR. Based on that technology, the .Net Micro Framework 1.0 was presented at the 2006 MEDC (Mobile and Embedded Developers Conference).

This framework neither does rely on an operating system, nor is it one. It is a lightweight bootable runtime for embedded development. The main advantage of it is its hardware independence. As opposed to other embedded development frameworks, the code of software developed using the .Net Micro Framework does not interface with the hardware directly. That way, software written for this framework can be run on any other device that runs the TinyCLR with the drawback that only 32 bit systems are supported yet.

As can be seen in Figure 1, which gives an overview over the architecture of that framework, a “Runtime Component Layer” exists which abstracts the “Hardware Layer”. A porting kit was released which contains the code of this hardware abstraction layer. This way, hardware vendors can easily adapt that code and port the .Net Micro Framework to their devices. [5]

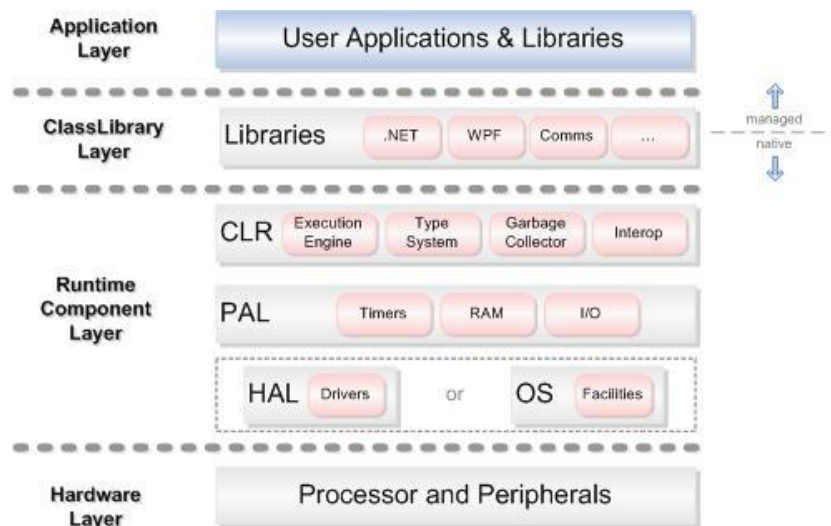


Figure 1 - Architecture of the .Net Micro Framework [6]

Though network communication has become easier with the class library provided by .Net Micro Framework there is still no built-in functionality which would allow for P2P communication. Developers need to use sockets or a vendor specific web service framework of Microsoft to transfer data between devices in a networked environment.

Back in 2008, four students of the Technical University of Vienna developed two reference implementations of a Space Based Computing Middleware for Java and the full .Net Framework which followed the XVSM (eXtensible Virtual Shared Memory) principle. [7] [8] [9] [10]

2 Objectives and Overview

The aim of this thesis is to develop a middleware for resource constrained devices based on the .Net Micro Framework which complies with a subset of the XVSM model on the one hand and suits the needs of embedded devices on the other hand. In the following it will be called “TinySpaces”.

The middleware shall follow the Extensible Virtual Shared Memory paradigm to remain compatible with other implementations of this kind, although providing less functionality.

The reason for this is that a peer-to-peer communication is very suitable for embedded devices, as their resource constraints often make it impossible to run as dedicated servers for several devices. Especially in scenarios where those devices are mobile and thus a centralized communication mechanism is hard to impossible to achieve, P2P communication, which is eased by the XVSM approach, is very useful.

It is also important for the middleware to follow the new formal model of an XVSM based middleware which was developed by eva Kühn et al [11]. The reason for the development was the observation, that the two reference implementations of XVSM (MozartSpaces and XcoSpaces) were not 100 percent compatible, although they were implemented at the same time and although the two development teams had a permanent interexchange.

Another reason for this formal model was the fact, that the two reference implementations were extensible by providing a possibility to add aspects, but their inner logic was not split into separate layers, which made it impossible to exchange a layer with some other implementation if needed. To achieve this modularity, a layered architecture was specified in the formal model.

This approach is very important for TinySpaces, as it makes it possible for developers to remove a layer that is not needed and would only stress processor and memory.

The first part of this thesis will provide an overview of the layered architecture of XVSM to give the reader the background information needed to understand the architecture of TinySpaces.

Next there is an introduction to the .NET Micro Framework along with its advantages and disadvantages to give an understanding why this technology was chosen to develop TinySpaces.

Furthermore networked embedded devices along with their common properties are presented and their requirements, which need to be met by TinySpaces, are explained.

The main part of this document puts an emphasis on the implementation of TinySpaces and explains which compromises need to be made to comply with the XVSM standard, or at least a subset, and also meet the requirements of resource constrained devices. In order to understand this thesis it is recommended to read the thesis of Stefan Craß [12].

Finally benchmarks on the performance and memory utilization of TinySpaces, and on the performance, packet-size and code-size of its three serialization mechanisms are made to determine whether the requirements of networked embedded devices are met.

3 Space Based Computing Middleware

“Space based computing (SBC) is an innovative and powerful concept for the coordination of autonomous processes. It is based on the notion of a common, abstract space connecting distributed processing entities over a network. Instead of explicitly exchanging messages between individual processes or performing remote procedure calls, processes communicate and coordinate themselves by simply reading and writing distributed data structures in a shared space.” [12]

To understand this definition, the terms “remote procedure call” and “exchanging messages” need to be explained in more detail.

1.1 Excursion: Remote Procedure Calls (RPC)

When using remote procedure calls to communicate over a network, a remote method of some service object is mapped to a local interface implemented by a proxy object. This object is then used locally to call the remote object [13].

One disadvantage of using RPC calls is that no location transparency is given as all communicating parties always need to know the physical addresses of each other. Another is that this communication always follows a client-server model.

Additionally RPC calls are often synchronous, meaning the calling client is blocked while the server processes the request.

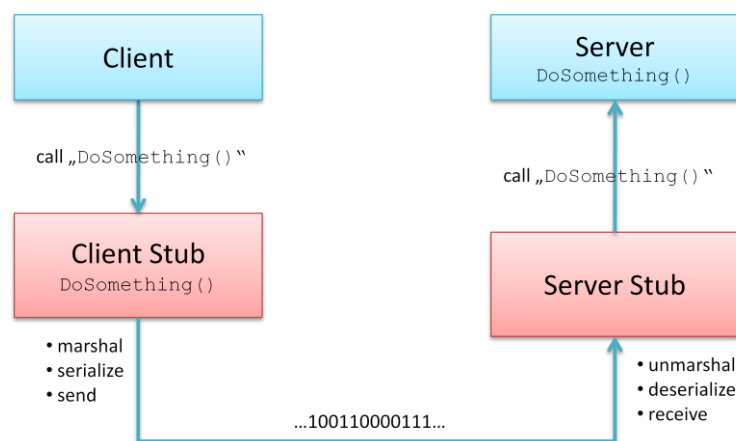


Figure 2 - RPC Communication

1.2 Excursion: Message passing

Using message passing for remote communication, a message containing all required information for a call (function invocation, data packets, security token etc.) is sent to the remote system. A very

prominent protocol based on this approach is SOAP (Simple Object Access Protocol), which is still widespread.



Figure 3 - Message Passing

The main difference between the two mentioned communication styles and space based computing is the fact, that space based communication is always stateful whereas remote procedure calls and message passing are stateless.

A modern stateful communication paradigm is message queuing.

1.3 Excursion: Message Queuing

Message queuing can be seen as the stateful approach of message passing. Before any communication can happen, message queues need to be installed which serve as intermediate stations. An advantage of this approach is that asynchronous communication can be achieved. This means that two communicating parties do not need to be running at the same time to exchange messages. The drawback however is that they are dependent on the message queues.

Additionally one-to-many communication can be achieved, but each receiving party needs its own message queue, as reading from a queue is always consuming the message. Therefore this approach is not very flexible and becomes cumbersome once multiple parties need to interact with each other.

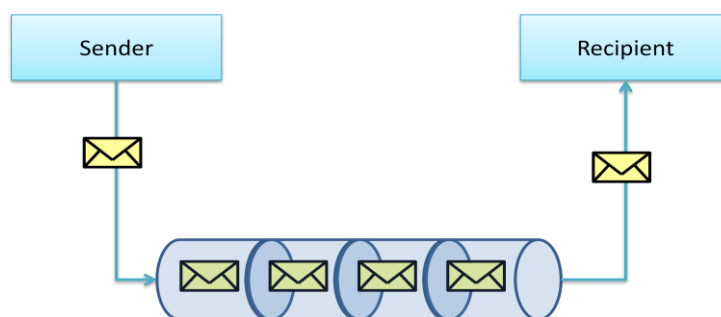


Figure 4 - Message Queuing

Communication with space-based technology is, as already denoted, always stateful, which means that data written into the space is not necessarily immediately removed. There may be several communication parties reading this data before one of them might finally remove it. Some of the readers might have also connected after the data had already been written into the space.

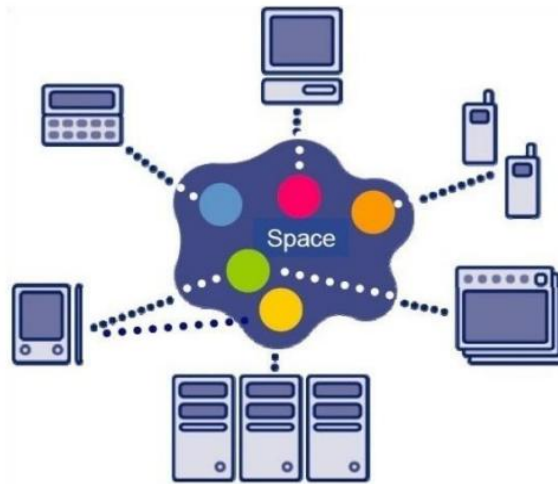


Figure 5 - Communication using a space

“Processes interact in a data driven way supporting event-driven architecture concepts. This leads to a decoupling of all participants concerning time, space and reference.” [12]

What this means is that many space-based middleware systems support some kind of publish-subscribe pattern which enable participants of the space to subscribe to certain events, for example for an entry written into the space that matches a specific pattern, and in the following get notified by the space if such an occasion happens.

A simpler approach would be to try to read from the space, and as the requested entry is not present at that time, the operation blocks until this condition changes.

An example for such a space-based computing middleware would be JavaSpaces [14] from Sun, but a lot more exist. For a survey of space-based systems we refer to the master thesis of Thomas Scheller where many of these middlewares are compared to each other. [7]

1.4 JavaSpaces

JavaSpaces is a well known space-based middleware implementation and based on the Linda model. Therefore all data that is written to the space has the form of a tuple, which is simply an ordered list of arbitrary elements (i.e.: $\langle \text{"xyz"}, [125], [1.5] \rangle$). Therefore, to read this tuple x from the space, only Linda tuple matching can be used. This means that an incomplete tuple y is sent to the space, which specifies what you are searching for. (i.e. $\langle \text{"xyz"}, [], [] \rangle$ – the empty brackets resemble wildcards)

The JavaSpaces runtime now searches for a tuple which matches all the defined parts of the sent fragment, except the parts that contain a wildcard, which are simply ignored.

This approach is simple and can be used to cover lots of scenarios, but it can become cumbersome in others. For example to realize a producer/observer scenario (writing and reading tuples in a “First In – First Out” manner) for publishing simple messages, an extra “count tuple” would have to be written into the space, which contains the current count of tuples belonging to the producer/observer scenario. This tuple is updated by each producer when it adds another message, and by each consumer when it removes one. The tuples that contain the actual messages additionally need to be numbered and

contain a fragment specifying which tuples belong to that “FIFO group” (here “po” for “producer-observer”). A scenario with three produced tuples containing simple messages is depicted in Figure 6.

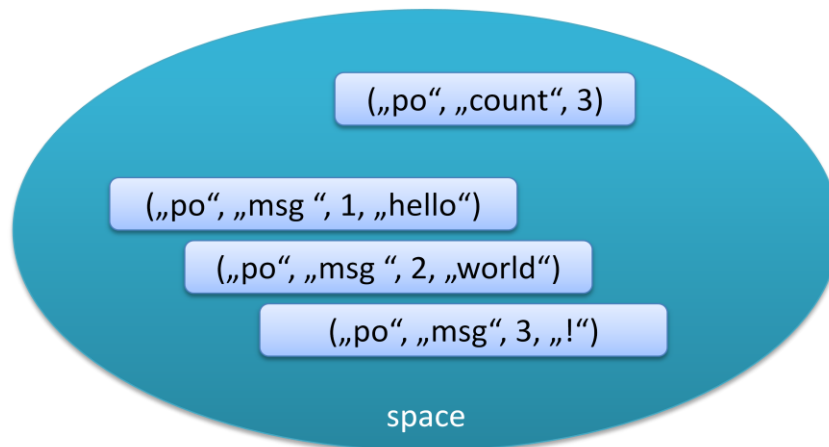


Figure 6 – Producer/observer implementation with JavaSpaces

If a producer needs to add another message, it first has to read the “count tuple” to determine the number (or index) of the message which should be added, which is four in this case. Then the “count tuple” is updated with the value of four, and the tuple containing that message is written with that value.

An observer needs to read the “count tuple” first, to determine how many tuples containing messages it has to retrieve. In the current example it would be four messages. Then it could make a read request separately for each message, or use a wildcard for the index value of the messages to retrieve all of them in one step. However, in the latter case they could be unsorted.

For more information on that topic we refer to the lecture of Eva Kühn [15]. For a detailed comparison of space based computing middleware and message queuing see [16].

1.5 XVSM

XVSM builds on top of the space-based computing paradigm, but offers additional coordination concepts and thereby goes beyond Linda tuple matching by introducing coordinators, containers and entries.

Entries are simply the data structures containing the actual user data. They are stored in containers, so never directly in the space itself. This allows a logical grouping of data and therefore reduces the risk of erroneously manipulating data. Additionally, an entry can store a reference to another container. Therefore even a hierarchical structure can be created. The last piece in this puzzle is the coordinator. A coordinator specifies at runtime how entries are read from and written to a container. A FIFO-Coordinator, for example, assures that the entry that has been written to a container first is also read first.

Therefore only container with a FIFO-Coordinator has to be created to realize a producer/observer scenario. The following figure (Figure 7) depicts how entries are written to and read from such a container.

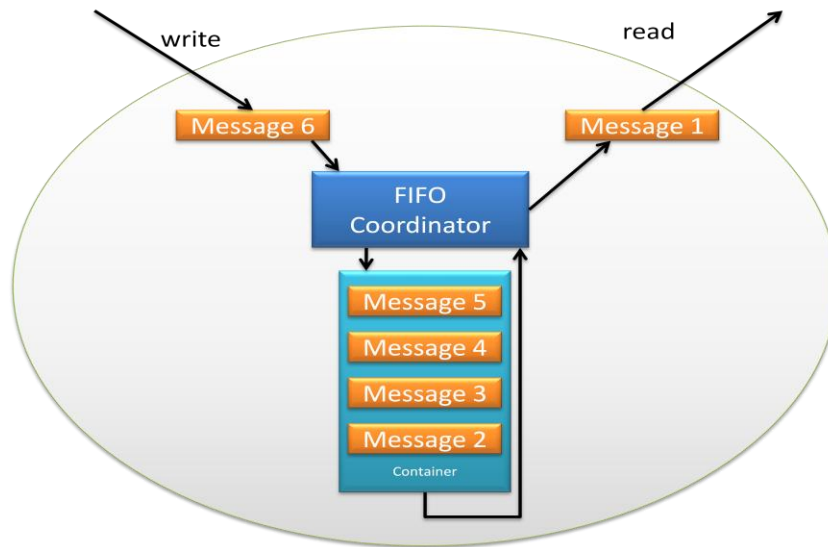


Figure 7 – Producer/observer implementation with XVSM

Using this approach a producer just has to write a message into this particular container and therefore does not have to manage the messages' indexes. An observer only needs to make a read request, which targets that container, to retrieve one or more messages in the correct order, as this is handled by the FIFO-Coordinator. For a more detailed description of coordinators in XVSM we refer to [10].

These building blocks show very well, how XVSM sets itself apart of other space-based computing middleware systems. Furthermore, it was shown in [17] that separating a space into logical containers and using custom coordinators suited for a specific scenario can also lead to significant performance improvements. This is another reason why XVSM was chosen, among other space based computing middleware specifications, to implement TinySpaces.

1.6 Layered Architecture of XVSM

To give the reader a basic understanding of the design of TinySpaces, this section provides a short overview of the layered architecture of XVSM. For a detailed description we refer to the master thesis of Stefan Craß [18] and [11].

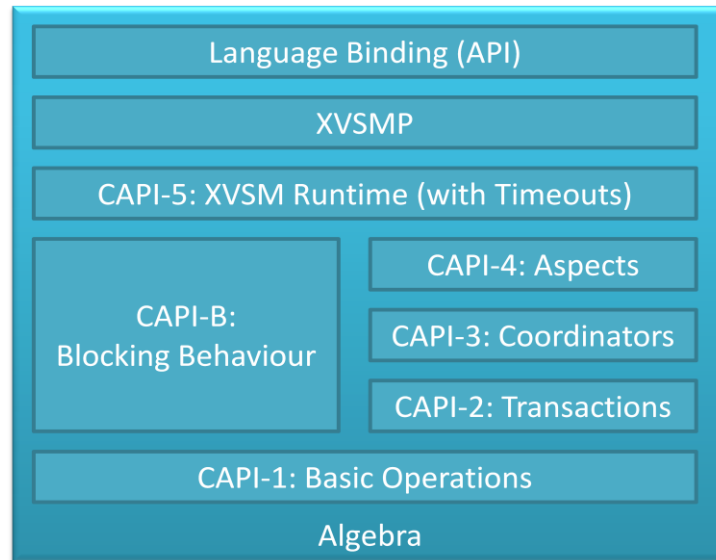


Figure 8 - Layered Architecture of XVSM

1.6.1 Algebraic Data Structures

XVSM defines several algebraic data structures to store information. **Properties** are formally name-value pairs that form the attributes of an object. **Metadata** is stored in properties, which are marked using special labels. Examples for metadata are the current count of entries within a container, the information whether an entry is locked by a transaction and which transaction holds the lock, et cetera. **Entries** formally consist of properties and are used to store user specific information in the space. Furthermore, one or more **containers** may exist in a space. They are used to store entries, therefore entries are not written directly into the space itself. This makes it possible to separate them into logical groups and even to create a hierarchy of containers. This approach reduces the risk of erroneously manipulating entries which do not belong to a logical group.

1.6.2 CAPI-1: Basic Operations

1.6.2.1 Atomic Operations

As the name implies, this layer consists of a "...set of synchronous operations on the XVSM algebraic data structures ... which either succeed or fail immediately." [11 S. 3]

These operations are

- **Read**, which is used to retrieve an entry/entries from a container without removing it/them,
- **Take**, which is used to read *and* remove an entry/entries from a container in one step,
- **Write**, which is used to insert an entry into a container.

These operations are called synchronously as they never block but either fail or succeed immediately. This is important as it eliminates the risk of race conditions. While only the operations Read and Take may return entries, all three of them deliver a status code to provide information about their outcome.

These defined codes in CAPI-1 are

- OK ... operation finished successfully
- NOTOK ... an unexpected error occurred
- DELAYABLE ... the operation did not finish successfully, but can be retried anytime. An example for this would be that a READ operation was called on an empty container.

This status code is delivered through all layers up to CAPI-5 where the corresponding request is eventually further processed.

1.6.2.2 Query Language

Additionally the built-in XVSM Query Language is defined within this layer. It provides additional expressiveness over selectors which are contained in CAPI-3. The standard queries, which are defined in [11], are:

- sortup(l)
... sorts the list of entries ascending depending on the value of the property with label "l".
- sortdown(l)
... sorts the list of entries descending depending on the value of the property with label "l".
- cnt(n)
... returns first n entries.
- distinct(l)
... returns a list of entries which are unique according to their value for the property with label "l".
- $x \in \text{range}$
... returns a list of entries, whose property lies within the specified range.
- $x \notin \text{range}$
... returns a list of entries, whose property does not lie within the specified range.

Queries can be piped to aggregate their logic and additionally extended by the user, by defining custom queries.

1.6.3 CAPI-2: Transactions

A transaction encloses a set of operations and allows for either completing all of them successfully, or none of them. The exact behavior is defined by the term "ACID" which, according to [19], stands for

- **Atomicity** .
This means that a transaction needs to either commit or rollback all of its corresponding operations in one single unit of work.
- **Consistency**

This means that after committing or rolling back an operation, the environment is always in a consistent state.

- **Isolation**

This “refers to the degree to which individual transactions interact with each other.” (Richards, 2006, S. 5) “Isolation is a function of consistency and concurrency. As the level of isolation increases, consistency increases and concurrency decreases.” [19]

- **Durability**

This refers to the fact that when an operation is committed, it is guaranteed that these changes are permanent.

CAPI-2 consists of the operations Read, Write and Take. These have basically the same syntax as the operations listed in CAPI-1, but require a running local transaction as additional parameter. Furthermore, they are non-blocking as well, but additionally introduce locking. This provides the possibility to get exclusive access to any XVSM algebraic data structure.

The CAPI-2 operations return the following status codes:

- **OK** ... the same as explained in section 1.6.2.1
- **NOTOK** ... the same as explained in section 1.6.2.1
- **DELAYABLE** ... the same as explained in section 1.6.2.1
- **LOCKED** ... this code is introduced in this layer and is returned if an algebraic data structures required for the CAPI-2 operation could not be locked, as another transaction already has a lock on it.

In XVSM basically two types of transactions exist.

User transactions are created, committed and rolled back explicitly by a client application using the API of XVSM.

Sub transactions are used internally to encapsulate a single operation. Thereby all operations need to be encapsulated by a transaction. If for example a user makes a request without specifying a transaction, the runtime layer implicitly creates one for that request. Furthermore, sub-transactions are used by the coordination layer to encapsulate a single CAPI-3 operation.

1.6.4 **CAPI-3: Coordination**

This layer contains the coordination logic which consists of coordinators and selectors.

1.6.4.1 Coordinator

A container may have one or more coordinators which are specified at container creation time. These coordinators are responsible for defining, how entries are written to and read from the corresponding container. For example when reading from a container using a FIFO-Coordinator the entry returned will be the one written into this container first.

Additionally, the coordinator specifies its own type of selector which is used to offer additional information about which entries should be read or destroyed, or how entries should be written into the container.

For example a Key-Coordinator works similar to a hash table, using unique keys to identify the entry that has to be read. When writing an entry with a Key-Coordinator, it needs to get the key that has to be used to identify this entry. This is achieved by adding a Key-Selector to the entry which contains the key value.

When an entry is written into a container the coordinators need to get informed about the actions taken to keep their accountant data in a consistent state. Therefore each coordinator specifies an accountant function which is called whenever an entry is read, written or taken.

For example, when an entry is being removed using another coordinator, the Key-Coordinator needs to be informed about that event to be able to remove the corresponding key.

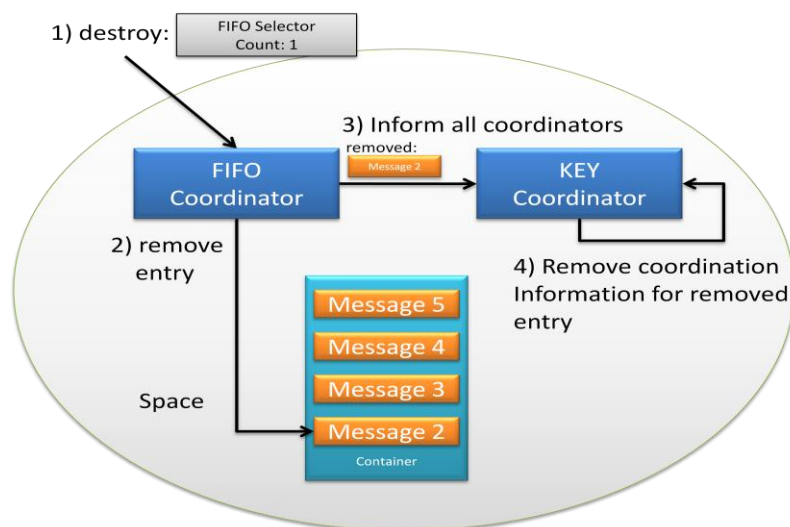


Figure 9 - Destroying an entry from a container with multiple coordinators assigned

1.6.4.2 Selector

A selector contains all the required information needed by a coordinator to succeed in reading or writing an entry/ entries. Each coordinator has its own type of selector.

For example a FIFO-Coordinator has a corresponding FIFO-Selector which can be used to specify the number of entries that should be read or removed from a container.

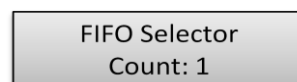



Figure 10 – FIFO-Selector

A LINDA-Coordinator on the other hand might specify the pattern that shall be used to identify a viable entry. The following figure drafts a LINDA-Selector which searches for a tuple with anything as first part (*null* is a wildcard), an integer with the value 12 as second part, and the string “vienna” as third part.

A rectangular box with a thin black border. Inside, the text "LINDA Selector" is on the top line, and "Pattern: <null,12,\"vienna\">" is on the bottom line.

LINDA Selector
Pattern: <null,12,"vienna">

Figure 11 – LINDA-Selector

1.6.5 CAPI-4: Aspects

The modularity achieved by separating the building blocks of XVSM into separate layers already allows for adapting and/or extending it. An example could be replacing the transaction layer that provides locking at entry level with another one that only allows for container level locking. That way concurrency would be reduced, but performance improvements could be achieved depending on the scenario the space is used in. However a lot of knowledge is needed to achieve this task.

Aspects in contrast allow for extending an existing space without having to re-implement a whole layer. In fact they are engaged to specified points in a space which are called insertion points or ipoints for short.

An example would be a security aspect, which assures that only authorized users can access the data in its space. [7 S. 98 ff.]

Currently XVSM has no complete formal definition of aspects although they have already been implemented in XcoSpaces and MozartSpaces. Therefore the description of aspects in [7] is used as a source here.

The ipoints where aspects can be added are:

- Pre-ipoint

Aspects engaged in this ipoint will be called before the corresponding operation is called. In the following those aspects will also be referred to as pre-aspects

- Post-ipoint

Aspects engaged in this ipoint will be called after the corresponding operation was called. In the following those aspects will also be referred to as post-aspects.

Additionally, more than one aspect can be added to a specific ipoint and in that case, the aspects are called sequentially. They can also affect the execution of following aspects and even the runtime behavior by returning one of the following status codes:

- OK

The aspect call was successful and the next aspect can be executed.

- SKIP

If this code is returned, all succeeding aspects for the current pre- or post-ipoint are not going to be called. Additionally, if SKIP was returned by a pre-aspect, the operation itself is skipped too. This can be used to replace a whole operation with an aspect.

- RESCHEDULE

When this code is returned, all succeeding operations and pre- or post-aspects are canceled and the request that caused this operation to be executed is rescheduled.

- ERROR

This status code indicates that the aspect does not want any succeeding aspects or the operation itself (if the code was returned by a pre-aspect) to be called - for example an aspect does not allow the current user to access a specific container - or that an unexpected error occurred within the aspect itself. The error is thereupon returned to the user.

Furthermore, there exist two types of aspects in XVSM:

Space aspects, as the name implies, space aspects work at space level, so they are global aspects.

Examples for ipoints are:

- Pre-/Post-CreateContainer
- Pre-/Post-CreateTransaction
- Pre-/Post-CommitTransaction
- Etc.

Container aspects only affect a single container and therefore are local aspects. Examples for ipoints are:

- Pre-/Post-Write
- Pre-/Post-Read
- Etc.

1.6.6 CAPI-5: XVSM Runtime (with Timeouts)

CAPI-5 contains the logic to schedule incoming and outgoing requests and responses. The current architecture, as introduced in [7], looks as follows:

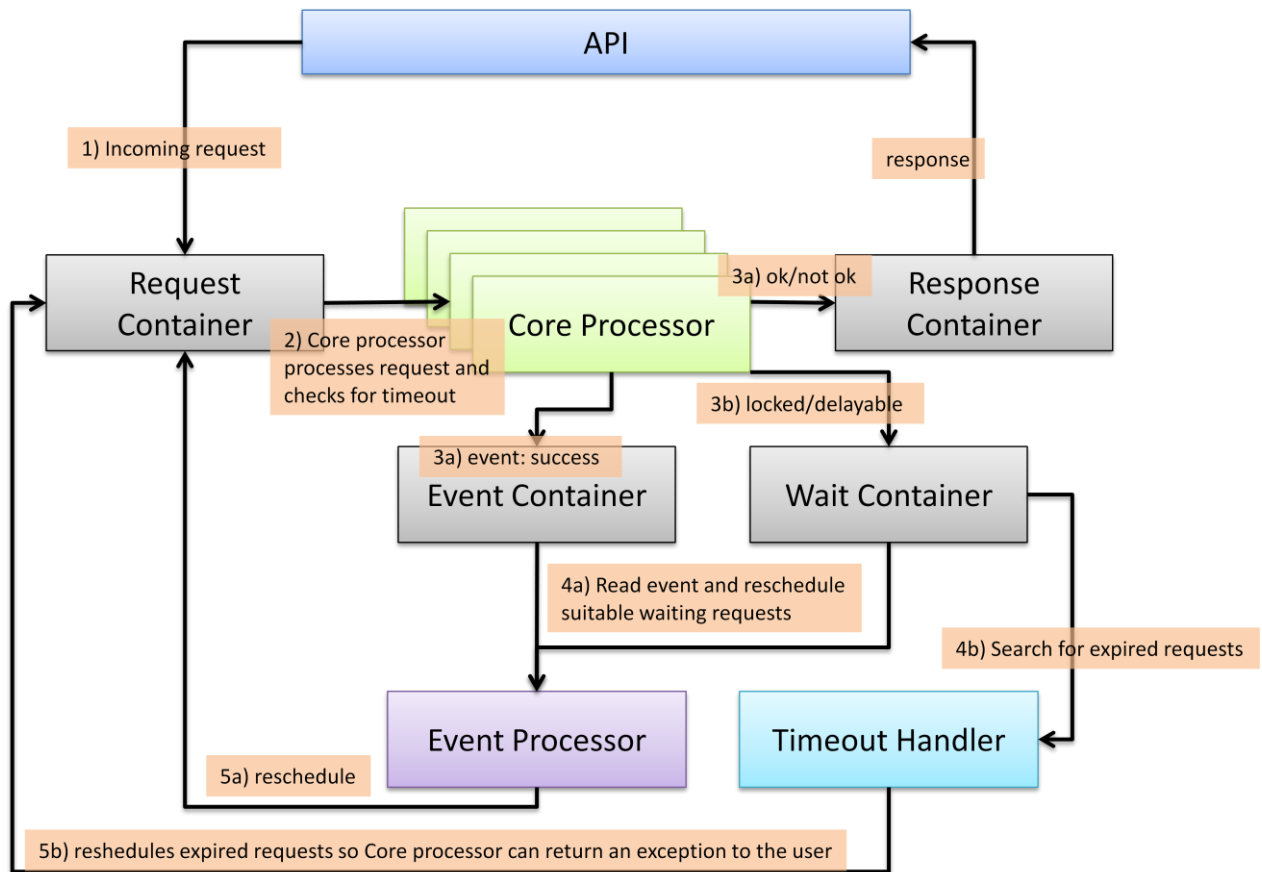


Figure 12 - XVSM Runtime

Requests, whether local or remote, are added into the request container, which is used by multiple core processors that take one request after the other and process them.

- If the operation belonging to that request succeeds (all operations return status code OK) or fails completely (NOTOK is returned), a response is created, written to the response container and an event is created and put into the event container.
- If the operation fails with the status DELAYABLE or LOCKED, the request is put into the wait container and needs to be rescheduled by the event processor as soon as any suitable event arrives.

The event processor waits for events fired by the core processor and uses their provided information to wake up waiting requests from the wait container. These requests are then again put into the request container.

A very important part of the runtime is the timeout handler. Each request provides a timeout value, which points out when the request expires. Whenever this happens the timeout handler removes the request from the wait container and reschedules it. The core processor which processes the expired request will notice that and create an error response, which is then written into the response container. This currently is the only way to resolve deadlocks.

1.6.7 **XVSM Protocol**

Currently the new formal model of XVSM does not define an XVSM protocol, as its development is still in progress.

Its task is to define how two heterogeneous implementations of XVSM communicate with each other.

However, XcoSpaces and MozartSpaces already provided a protocol based on an XSD Schema allowing them to collaborate, although they were based on different technologies (.NET and Java). At first, only primitive data types could be send from one space implementation to another, but in 2008 the TupleConverter module was implemented, which made it possible to send arbitrary objects (i.e.: classes) from XcoSpaces (.NET) to MozartSpaces (Java) and vice versa by simply decorating them with Attributes (.NET) or Annotations (Java). For more information on the original implementation of the protocol see [8 S. 61ff.] and [10 S. 73 ff.]. For more information about the TupleConverter see [20].

1.6.8 **Language Binding (API)**

The language binding acts as a facade to the client application which in turn uses the API to leverage the functionality of XVSM and never directly interfaces with any of the underlying layers. This is very important, as layers could easily be replaced without having to recompile or even adapt the client application.

Currently it is not defined how the API should exactly look like and there are multiple ways to implement them. For example, the current implementations of XVSM (XcoSpaces and MozartSpaces) provide a synchronous API (except of the part for notifications), but an asynchronous will also be implemented in future XVSM versions.

2 Embedded Development and the .NET Micro Framework

As already denoted in the introduction, the .NET Micro Framework was chosen to implement a XVSM based middleware for embedded devices. This section provides detailed information about the framework as well as about embedded systems to help the reader to understand this decision.

2.1 **Embedded Systems**

“Embedded systems are information processing systems that are embedded into a larger product and that are normally not directly visible to the user.” [21 S. 1]



Figure 13 - USBizi by GHI Electronics¹Error! Hyperlink reference not valid.

These systems consist of one or more embedded devices and generally share common characteristics as denoted in [22]:

- They are usually connected to their physical environment using
 - **Sensors** to collect information about it (i.e.: a temperature or humidity sensor)
 - **Actuators** to control it (i.e.: a servo which closes a valve)
- They frequently are safety-critical and therefore **need to provide dependability**.
Dependability encompasses the following aspects:
 - reliability
 - maintainability
 - availability
 - safety
 - security
- As embedded systems usually consist of resource constrained devices, they need to be **efficient**. Metrics to measure efficiency are:
 - Energy consumption
 - This is important as many systems are mobile and use batteries as power supply.
 - Code-size
 - All code to be run on a device has to be stored, but there are typically no hard discs available which would offer large amount of storage space.
 - Code-size also affects energy consumption as can be seen in [22].
 - Run-time efficiency
 - This means that the software should use the minimum amount of resources such as CPU and memory.
 - Cost
 - For high-volume systems it is important that the separate hardware modules are cheap.
- These systems are **dedicated to one specific application** and will never run any other software at the same time like personal computers.

¹ <http://www.ghielectronics.com/images/extras/USBizi-SIZ-large.JPG>

- They are often **hybrid systems** meaning they consist of digital and analog parts.
- Embedded systems typically are **reactive systems** meaning that they wait for a specific input, perform a computation and generate a new output. [23]
- Typically must **meet real-time constraints**. This means that not computing the results within a given time frame would cause in a serious loss of quality.
 - A time-constraint is called **hard** if the corresponding system would completely fail its task if it is not met, according to [24], which could result in a catastrophe. For example the antilocking break system could cause a catastrophe if it would not react to a critical break maneuver of the driver.
 - Otherwise it is a **soft** constraint.

2.1.1 Application Areas of Networked Embedded Systems

Over the time there has been an increasing count of application areas of embedded systems. This section puts an emphasis on the application areas of networked embedded systems. The information provided is taken from [22].

Networked embedded systems (NES) usually consist of multiple distributed embedded devices collaborating via field area networks. The benefits are numerous, including increased flexibility, scalability, maintainability and extensibility. Usually field area networks tend to have low data rates a small size of packets, but data rates above 10 Mbit/sec are becoming more common. Additionally, these networks often require real-time capabilities.

Such systems again are used in different application domains where different functional and non-functional requirements are imposed.

The area of industrial automation is the origin of field area networks and started in the end of the 1960s in the nuclear instrumentation domain. Examples are systems which control a nuclear power plant or are used in avionics applications. As these systems often are safety-critical they require real-time capabilities.

In **building automation** NES are used to control the internal as well as the immediate external environment of buildings like office buildings, or shopping complexes and are also emerging into the area of industrial buildings. Their main services typically include control of climate, heating, artificial lights, power, gas, water supply, et cetera. As opposed to systems in the industrial automation domain these systems hardly ever require hard real-time communication.

Nowadays **automotive NES** gain importance as more and more collaborating embedded devices are built into automobiles to replace mechanical components in order to increase security. They have already become an intrinsic part of this domain. Examples for vehicle functions controlled by these devices are electronic engine control, active suspension or antilocking break systems. These systems always enforce hard real-time communication.

Sensor networks typically are self-organizing and consist of devices, which are for example embedded in the ecosystem, and communicate using wireless media. In addition, they usually have to

be highly available, provide low and predictable delay of data transfer and support a high number of sensors and actuators as well as low power consumption.

2.2 The .NET Micro Framework

As already denoted, the .NET Micro Framework is a bootable runtime environment for resource constrained devices.

Its main aim is to ease the development process in the area of embedded devices and thereby radically reducing costs and fasten time to market by providing the following features, which set it apart from other traditional embedded platforms:

- Introduces managed code to embedded devices

This means that the generated code is executed in a virtual machine which “manages” the execution. One resulting benefit is that resources need not to be freed manually when they are not used anymore as this is done by a garbage collector.

- Supports the modern programming language Visual C#

Visual C# offers a high level of abstraction and therefore is very easy to use. It is syntactically similar to Java.

- Allows for developing platform independent and fully reusable code

As the developer, when using the .NET Micro Framework, writes code which is executed by a virtual machine, this code is absolutely independent from any hardware and can therefore be easily reused.

- Managed Drivers

The framework treats hardware components as objects by abstracting hardware access. This makes it possible to program hardware components by setting the properties of the corresponding object instead of dealing with hardware details like setting bitmasks to configure a component. Therefore even this code is independent from the underlying hardware.

- Supports the development and use of hardware emulators

The ability of the framework to abstract from the underlying hardware allows for running the environment within an operating system like Windows Vista. This makes it possible to develop emulators for embedded devices which can be used to develop software for that device that is not physically present. The framework even provides reusable types which can be used to easily implement a new emulator.

- Includes Visual Studio support

Visual Studio is a widespread integrated development environment for .NET platforms. It can be used to develop applications using the .NET Micro Framework and even to debug applications while they are running on an embedded device or within the emulator.

- Provides a base class library

The base class library contains a collection of reusable types which reduce the amount of code which has to be implemented for an application by providing out of the box functionalities. As most part of this library is a subset of the full .NET Framework and therefore compatible, it is possible to share classes developed for the .NET Micro Framework with all other .NET platforms.

One major drawback of the .NET Micro Framework is, however, that it cannot be used in environments where real-time capabilities are required.

Since the launch of the .NET Micro Framework version 4 on Nov 16th, 2009, the framework became open source together with the full porting kit. Almost all of its components are published under the Apache 2.0 license. [25] More information can be found in [26].

These features turn the .NET Micro Framework into a highly productive platform which has already been proven by several surveys. The home control manufacturer Leviton Manufacturing for example was able to reduce the time to market of a product to only three months while additionally reducing hardware and licensing costs. [27]

As opposed to this, according to what Colin Miller, program manager of the .NET Micro Framework, mentioned in an interview, the average time to market in the embedded sector is about eighteen months. [28]

Another vendor which achieved better productivity using this framework is Inthinc as Corey Catten, CTO of Inthinc, stated in an interview: "When creating solutions that work to save human lives, quality is the prime directive. The .NET Micro Framework helped to reduce hardware costs, streamline the development and add a new level of flexibility to the entire process" [29]

The following section describes how the .NET Micro Framework achieves this level of abstraction using a layered architecture.

2.2.1 The Layered Architecture of the .NET Micro Framework

As the figure below shows, the .NET Micro Framework is separated into the following layers.

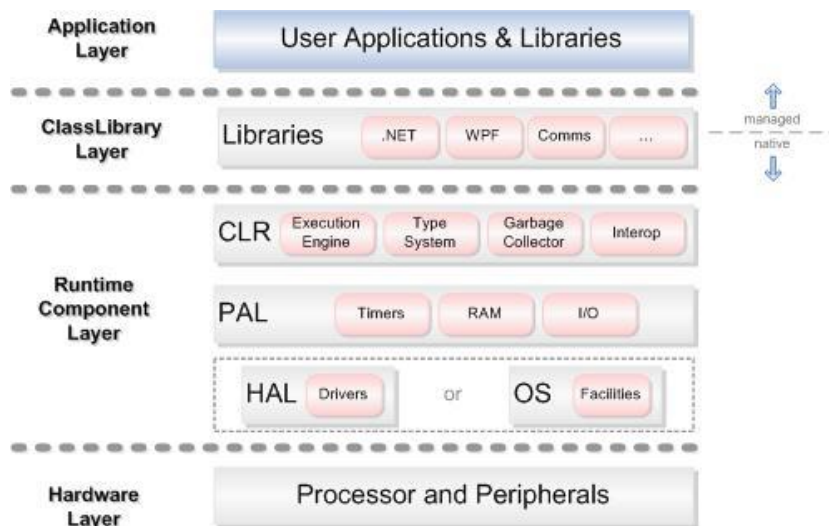


Figure 14 - Layered Architecture of .NET Micro Framework [6]

The **Hardware Layer** contains the hardware of the used platform. Currently the framework is supported by several processors which are based on the architectures of ARM7, ARM9, Cortex, XScale, ARC, and ADI Blackfin.

As can be seen in Figure 14, the lowest sub-layer of the **Runtime Component Layer** is subdivided into two layers:

The **HAL** (hardware abstraction layer) is used to interface the hardware layer directly and therefore contains C++ driver functions.

The **OS layer** (operating system) on the other hand can be used to run the .NET Micro Framework within an operating system. This is a great benefit, as emulators can be written and therefore devices can be developed and tested without even using any hardware.

The **PAL** (platform abstraction layer) is similar to the HAL with the only difference being that the driver functions of the PAL are independent from the underlying hardware.

The **CLR** (Common Language Runtime) of the .NET Micro Framework is slimmed version of the .NET Framework CLR and therefore is often referred to as Tiny CLR. It is responsible for executing the IL (intermediate language) code which is generated by the C# compiler.

The **Class Library Layer** contains a collection of reusable object-oriented types and can be seen as a subset of the class library provided by the full .NET Framework as for example only a very limited amount of data structures exists and most of the components provide reduced functionality. As can be seen in the figure above, this layer contains both, managed and native code. The reason for this is that time critical functionality is implemented using native code, as it is executing much faster.

2.2.2 Alternatives to the .NET Micro Framework

Although the main advantage of the .NET Micro Framework is its hardware independence, this ability is also a disadvantage in other cases. The reason for this is that code written to run on the .NET Micro

Framework needs to be interpreted at runtime by the Tiny CLR to achieve this independence and this in turn costs a considerable amount of performance.

Therefore for some application areas with hard real-time constraints, using native code is still the only choice. Hence, the programmer needs to use several different C dialects (depending on the platform), C++ or even assembly language.

As portability of the developed code became highly important in the last years MD(S)E (model driven (software) engineering) and MDA (model driven architecture) [30] has become a wide-spread approach. However, the corresponding tools are often modified to suite a specific scenario and do not generate the full source code, but only support partial- or skeleton generation, also because the modeling languages like UML (Unified Modeling Language) are not expressive enough. Thus the developers still have to write code manually after the actual code generation. However research is done on tools capable of generating the full source code by adding script languages to model the details the modeling language does not support as described in [31] just to give an example. In [32] a step further is taken by introducing a model-*integrated* approach. However, those tools are still under development and subject to research and are therefore not completely mature.

Additionally, several other platforms of Microsoft exist. The .NET Micro Framework differs from them as it "... is a bootable runtime module that brings the advantages of .NET programming to devices too resource-constrained to run other Microsoft embedded platforms." [5 p. 1]

To give an in depth understanding, the following section explains where the .Net Micro Framework fits.

2.2.3 Other Microsoft embedded platforms

Microsoft already provides a variety of embedded platforms and each one fits into a specific environment. The following table lists those environments, enumerates examples for each and indicates which XVSM implementation (TinySpaces or XcoSpaces) could be leveraged with them.

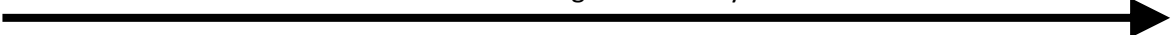
TinySpaces	XcoSpaces	XcoSpaces
.Net Micro Framework (managed code only)	Windows Embedded CE (managed code with the .NET Compact Framework)	Windows XP Embedded (managed code with the full .NET Framework)
<ul style="list-style-type: none"> • Auxiliary Displays • Sensor Nodes • Health Monitoring • Remote Controls • Robotics • Wearable Devices • Dataloggers • Home Automation • Industry Control • Vending Machines • ... 	<ul style="list-style-type: none"> • GPS Handhelds • Automotive • PDAs • Smart Phones • Dataloggers • Set Top Boxes • Portable Media players • Gateways • VoIP Phones • ... 	<ul style="list-style-type: none"> • Retail Point-of-Sale • Windows-based Terminals • Medical devices • Entertainment devices • Kiosks • ...
Increasing functionality 		

Table 1 - Where the .NET Micro Framework fits. [2 S. 5]

3 Related Work

There are many publications on embedded systems middleware. An important aspect of these middlewares is their real-time capability, which is needed in many areas of embedded systems and is especially hard to achieve in networked environments. A standard for end-to-end QoS (Quality of Service) of CORBA was introduced in [33], which makes it possible to testify when certain requests are processed in the worst case, et cetera. Currently, there are many middlewares which are based on CORBA like for example TAO [34] and emORB [1] which targets mobile embedded devices and OCP [35], [36] which is an open middleware specification and has already been used to control UAVs (unmanned aerial vehicles) and UACVs (unmanned combat air vehicles).

An area where XVSM is very well suited is the area of Industrial Automation. As can be seen in [17] XVSM can be used to efficiently coordinate multiagent systems in the production automation domain. This is an area where TinySpaces could be used to reduce hardware costs and energy consumption depending on the required features and workloads. Another application area is the one of Intelligent Transport Systems as is described in [37]. However, as the current XVSM specification does not provide real-time capabilities, many aspects of ITS (Intelligent Transport Systems) like accident- and traffic jam warning, just to give an example, cannot be provided at the moment. While the former publication concentrated on the communication between a central administration and moving vehicles, the EMMA (Embedded Middleware in Mobility Applications), which is introduced in [38], additionally aims to facilitate vehicle-to-vehicle communication and introduces wireless sensor networks to the automotive domain. While creating an XVSM based middleware capable of competing against EMMA would go out of scope of this thesis, it is thinkable that a space-based approach could be more suitable for sharing information between moving vehicles and thus should be subject to future research.

Currently a lot of research is done in the area of sensor networks. Sensor motes, which are sensor devices participating in a sensor network, need to be as efficient as possible as they are battery powered most of the time and need to provide the highest lifetime possible to reduce maintenance cost. For this reason TinyOS [39], which is a slimmed operating system for sensor networks, has been developed along with a specialized programming language for networked embedded systems which is called "nesC" [40]. There also exists a framework called TinyGALS which makes it easy to generate a slimmed and specialized operating system for that area to reduce code-size and lower energy consumption even more [41]. Furthermore, a query processing framework for such networks called TinyDB was created which can be used to retrieve sensor information using simple SQL-like (Standard Query Language) queries [42].

Due to this specialization, a much lower code-size and higher run-time efficiency can be reached. For example motes with a 5 MHz CPU and several kilobytes of ram exist. As the smallest .NET Micro

Framework device has a 55 MHz CPU (32 bit), it is clear that those devices are not suitable for sensor networks and therefore this thesis does not aim at providing a XVSM based solution for this area.

However, the combination of that technology with a space-based approach in the ITS area could be used to provide sensor information about the environment to approaching vehicles and a central administration, but that would be out of the scope of this thesis as well.

In the sector of building automation a middleware was introduced in [43], which is based on UPnP. However its aim is to interconnect and orchestrate heterogeneous domotic services to ease everyday life and not to coordinate multiple mobile agents, for example.

Energy aware programming is very important for developing networked embedded systems which is also shown in [42]. In general, those devices need to be reactive and not use busy waiting. This is a very important aspect for notifications, where as little network bandwidth as possible is to be used. Therefore it might be useful for some scenarios to enhance the current profile of notifications, which only supports topic-based notifications as described in 4.5.3, to also support content-based notifications which is described in [44].

The following figure depicts how the related work is structured.

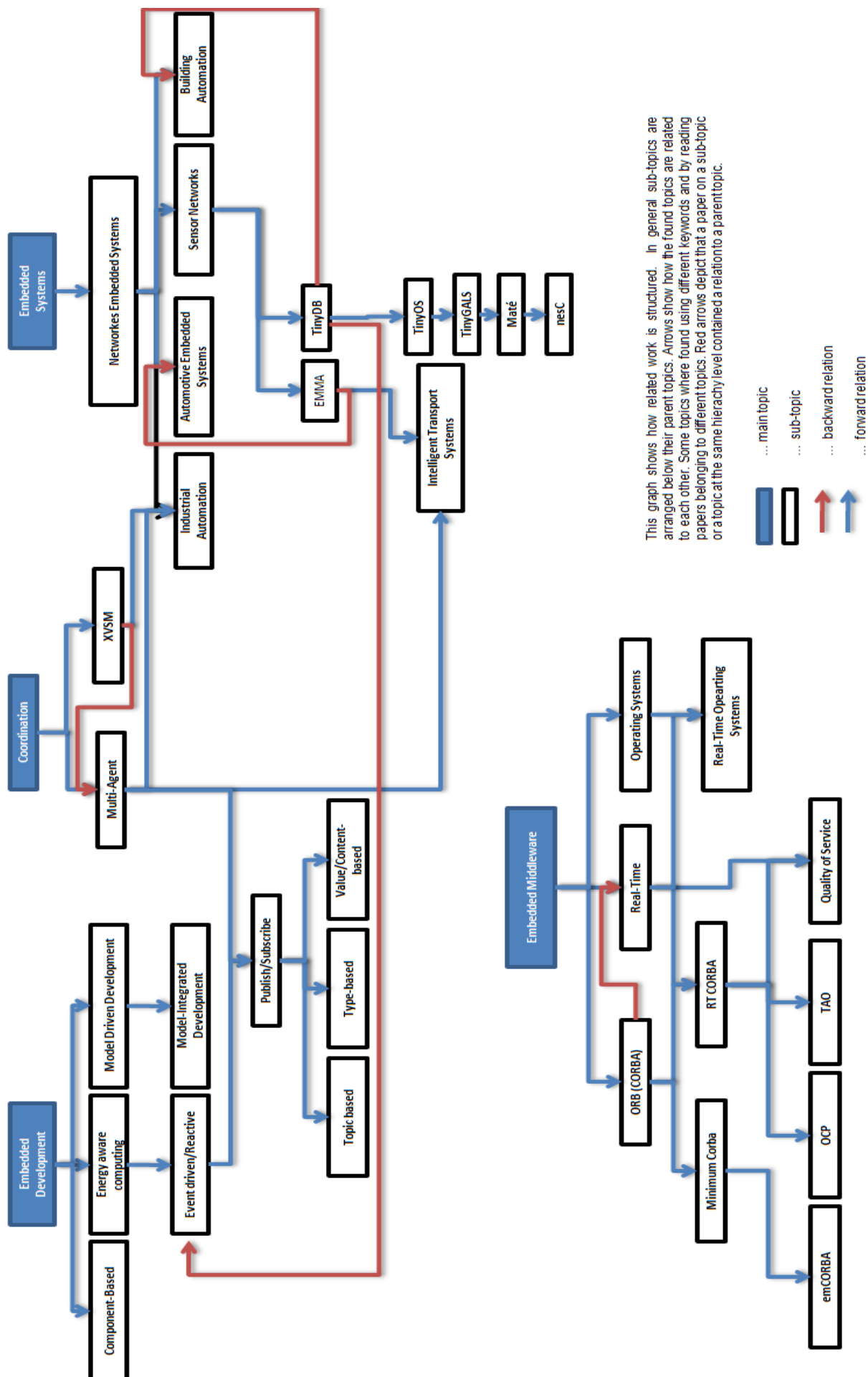


Figure 15 - Related Work Graph

4 TinySpaces Design

As the previous chapters aimed to provide background information about embedded development as well as the XVSM paradigm, this chapter explains how all this information was put together to develop TinySpaces. At first some general decisions are explicated followed by a precise explanation of each of the layers of TinySpaces from bottom up.

Note that some of the decisions were made by the XVSM Technical Board (abbreviated TB). This is a periodical meeting of colleagues, which are researching the XVSM technology. Therefore the writer of this thesis will refer to the TB whenever appropriate.

Additionally, the differences to XcoSpaces and the formal model of XVSM are pointed out whenever appropriate. XcoSpaces were preferred to MozartSpaces because the former is, as well as TinySpaces, based on a .NET platform and is therefore more adequate for a comparison.

4.1 Contract First Design in TinySpaces

A very important requirement for TinySpaces is to provide a level of modularity which allows for replacing whole layers if needed. Therefore a contract first approach was chosen.

From the view of contract first design software consists of components, which can be seen as logical units. Usually these components are separated in one or more assemblies, grouped by similar functionality.

In traditional software development client components directly depend on one or many other server component(s). This however makes it impossible to replace a component if needed without rebuilding the whole software.

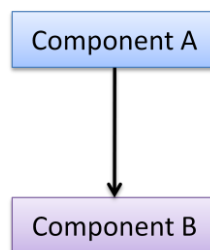


Figure 16 - Direct dependency between components

To overcome this issue contract first design introduces contracts which define how the corresponding components can be interfaced. Therefore the former client component is no more directly dependent on the server component(s) but only on the contract.

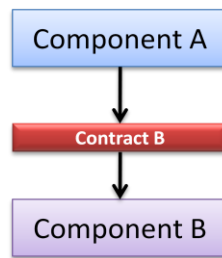


Figure 17 - Component interfacing contract

This allows for replacing the corresponding server component(s) without having to recompile the software. The new component only has to comply with the contract.

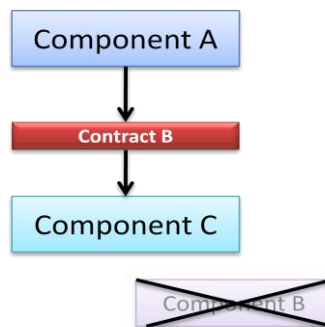


Figure 18 - Replacing a component using Contract First Design

In modern software development dependency injection frameworks like SPRING [45] exist which can be leveraged to wire up components at runtime, but this approach is far too costly for embedded devices as the resulting dependency graphs can become very complex [46]. In many cases dependency injection is configured using XML-based configuration files to provide configurability. XcoSpaces for example use that approach [7 S. 75 ff.]. For embedded devices this is not applicable as they typically do not provide any hard drive where this information could be stored on. Therefore this is often done at compile time.

For that reason TinySpaces requires a configuration object to be provided at startup, which contains the instances of the facades of CAPI-1 to 5.

4.2 CAPI-1: Basic Operations

In this first chapter about the implementation of TinySpaces, an overview of CAPI-1 layer is given. At first the corresponding contracts are explained followed by additional information about the current implementation.

4.2.1 Contracts

The following figure shows the main contracts needed to be complied with by any implementation of this layer.

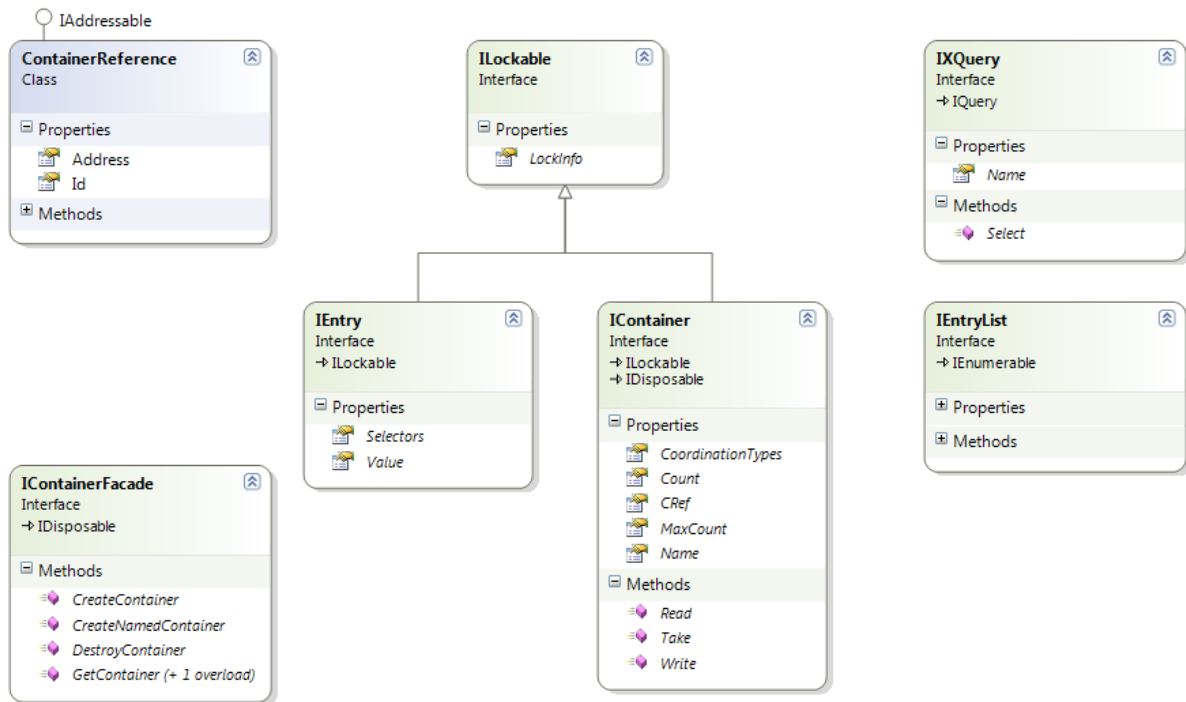


Figure 19 - Contracts of CAPI-1

4.2.1.1 ContainerReference

The ContainerReference class contains the information needed to access one specific container instance in the XVSM universe.

From the view of the formal model, a ContainerReference is a simple string. In XcoSpaces it is a combination of an identifier, represented by a UID, and an address represented by a simple string.

For TinySpaces a slightly different approach was chosen: At first, the identifier is a 32 bit unsigned integer. While a UID has the advantage that its value is always absolutely unique, generating those values is costly and the values require considerable more memory and bandwidth: While a UID is binary represented by 16 bytes, a 32 bit integer only uses 4 bytes. The drawback of using an integer value as identifier is that this value is only unique within the space that has generated it. For this reason the address value has to be taken into account to fully identify a container.

Moreover, the address is represented by a separate object, which provides the information about the transport protocol to use and the address value. This decision was made as parsing a string to get the protocol and address information, as it is done in XcoSpaces, is too resource intensive for embedded systems.

ContainerReference		
	Address	Gets/Sets the address of the space this container resides in.
	Id	Gets/Sets the locally unique ID of the container.

Table 2 - CAPI-1: ContainerReference

4.2.1.2 IContainerFacade

The IContainerFacade interface defines the methods needed to create, destroy and access a container in the local space.






 IContainerFacade		
	CreateContainer	Creates an unnamed container with a locally unique ID
	CreateNamedContainer	Creates a named container with a locally unique ID
	DestroyContainer	Destroys the container identified by the ContainerReference given.
	GetContainer	Returns the container identified by the ContainerReference. Throws a ContainerNotFoundException if no container was found.

Table 3 – CAPI-1: IContainerFacade members

4.2.1.3 IContainer

In contrast to the IContainerFacade interface, IContainer specifies how entries are written to- and read from a container.










 IContainer		
	CoordinationTypes	Contains the coordinators assigned to this container.
	Count	Returns the current entry count.
	CRef	Returns the corresponding ContainerReference.
	MaxCount	Returns the maximum entry count allowed.
	Name	Returns the name of this container.
	Read	Takes an array of IXQuery objects as parameter and uses them to select the appropriate entries which are then returned.
	Take	Takes an array of IXQuery objects as parameter and uses them to select the appropriate entries which are then returned and removed from the container.
	Write	Adds an IEntry to the container.

Table 4 – CAPI-1: IContainer members

As can be seen there is no “Destroy” method defined according to the formal model of XVSM. A good reason for this is that “Destroy” and “Take” differ only in the fact that “Take” returns the entries which are removed from the container, whereas “Destroy” does not.

However the entries removed from the container need to be propagated to the transaction layer (CAPI-2) as they are used as meta-information. For example a transaction that is committed gathers, among others, all entries that were removed and provides this information to the runtime layer (CAPI-5). The way this information is then consumed is explained in 4.6.2.4.

Furthermore, the IContainer interface directly provides access to its meta-information (Count, MaxCount, Name and CoordinationTypes). The drawback of this approach is that these pieces of

information cannot be locked and accessed exclusively and therefore cannot be retrieved by a user application using the XVSM API. Therefore it is not formally correct.

In XcoSpaces all metadata is stored in a separate container with a key coordinator. This satisfies the formal model, but would be too resource intensive for embedded devices.

4.2.1.4 IEntry

The IEntry interface is implemented by any class which is written into a container. It consists of two properties only. Any data the user application wants to write into the space is stored in the value property. Additionally, the entry contains an array of selectors as the single metadata. These are used by the coordination layer as some coordinators, as for example the Key-Coordinator, need additional information to read/write/take entries. The role of selectors is further described in 4.4.1.3.




 IEntry	
 Value	Gets/Sets the value of the entry. (can be any arbitrary object)
 Selectors	Gets/Sets the selectors belonging to the entry.

Table 5 – CAPI-1: IEntry members

4.2.1.5 ILockable

The ILockable interface is implemented by both, the IContainer and the IEntry interface, and provides a single property which can be used by the transaction layer (CAPI-2) to store arbitrary locking information for that entry or container. It therefore complies with the formal model described in [18], which specifies that meta-information can also be stored at container- and entry level.



 ILockable	
 LockInfo	Gets/Sets an arbitrary object and is used by the transaction layer to get exclusive access to an entry or container.

Table 6 – CAPI-1: ILockable members

The way the current implementation of the transaction layer of TinySpaces uses this property is described in 4.3.2.1.

4.2.1.6 IXQuery

The final interface in this layer is the IXQuery interface. It defines how any query used to access this layer has to look like.




 IXQuery	
 Name	Returns the name of the query.
 Select	Takes a list of entries and returns another which contains selection of those.

Table 7 – CAPI-1: IXQuery members

Although .NET in general provides the use of Delegates [47], which are a kind of managed function pointers, the decision was made to use objects inheriting from IXQuery as queries instead. The reason

for this is that a lot of queries need to store context information and thus are stateful. For example a Cnt query (see 4.2.2.2.1) needs to remember the number of results it has to return and queries created by the coordination layer (CAPI-3) need to have access to their coordinator's accountant information to perform their task. However, this would not be possible using delegates as they are stateless.

4.2.2 Implementation

In this chapter further information about the implementation of CAPI-1 is given.

4.2.2.1 Container

The current implementation of the IContainer contract in TinySpaces provides access to the contained entries in an atomic manner thus complying with the formal model. This means that only one operation (read/take/write) may execute at a time, thus reducing concurrency. But this approach also brings a major benefit:

Consider several execution threads which invoke those operations and try to take the same entries. The take-operation *o1*, which executes first, needs to execute several queries (like a Cnt query and a query generated by a Key-Coordinator) to get the final result set of entries. If *o1* was not executed in an atomic manner, a second take-operation *o2* could execute at the same time on a different thread. Supposed *o2* has only one simple query (like a Cnt query) to be run it could finish before *o1*, thereby taking entries *o1* is also going to take. However, entries can only be taken *once* and therefore this behavior would not be valid. This conflict is depicted in Figure 20.

Moreover, this would mean that the more queries an operation has to perform, the higher is the risk that it will fail. Accordingly, those operations would be disadvantaged.

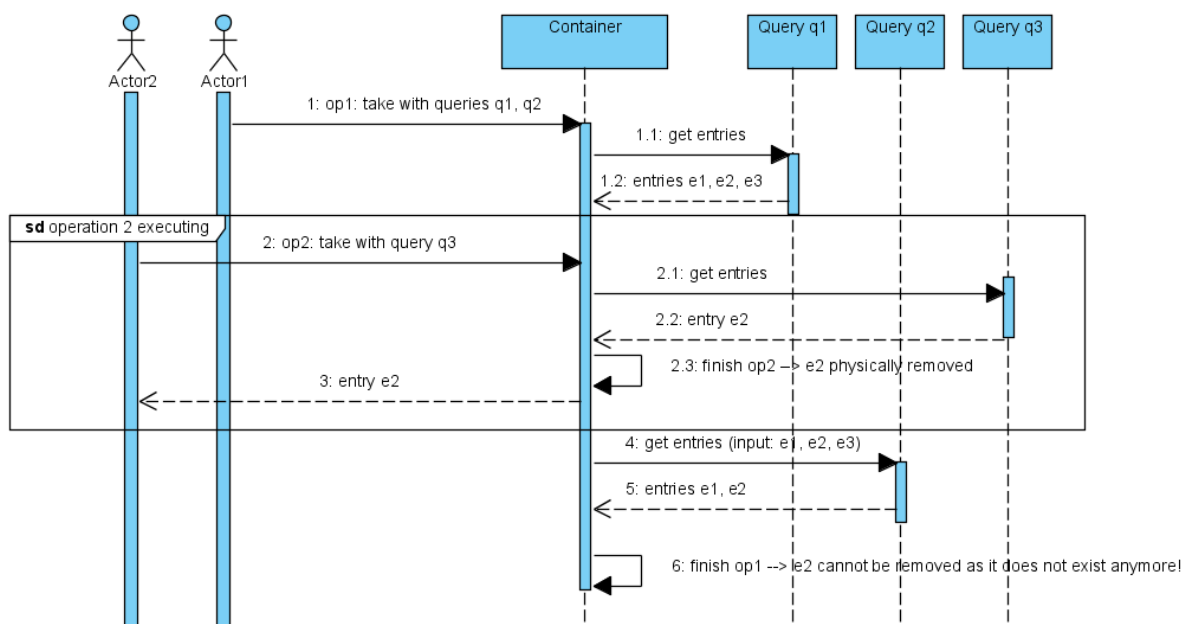


Figure 20 - CAPI-1: Concurrent operations

Another task of the container is to handle errors occurring in queries. Whenever any of the queries does not get enough entries to succeed for example, it throws a `DelayableException`. This exception is then caught by the container which in turn returns the status “DELAYABLE” and no entries to the upper layer (CAPI-2). Additionally, exceptions can be thrown by custom queries which have been erroneously implemented. In that case the container returns the status “NOTOK”.

4.2.2.2 Queries

As already denoted there are six queries predefined by the XVSM model. However, only the `Cnt` query could be implemented using the .NET Micro Framework. This is because that query is the only one which does require labels or even “label paths” to be specified, which identify the property of the entry or its inner “XTrees” that has to be taken into account like for example the queries “sortup” and “sortdown”. The only viable approach to implement these other queries would be to parse the provided label/label path (e.g.: “Value.Temperature” where “Value” and “Temperature” are two distinct properties concatenated to a label path) and use Reflection [48] to navigate through the type hierarchy and access the values of those properties. However, Reflection in .NET Micro Framework (version 4.0) does not allow for accessing properties of objects dynamically at runtime. [48] Additionally, this would cause too high computational costs.

In the following the two queries which are provided by TinySpaces out of the box are described.

4.2.2.2.1 Query: Cnt

The `Cnt` (which stands for “Count”) query takes a list of entries and returns another which contains the first *n* entries of the input list.

For example using this query with the value 1, only the first element of the list of entries provided by the foregoing query is returned.

If not enough entries are supplied, the query throws a `DelayableException` which is then handled by the container as already denoted (see 4.2.1.3).

4.2.2.2.2 Query: Reverse

This query is not defined by the XVSM model. However, it turned out to be very handy in some cases. The `Reverse` query takes a list of entries and returns a new one, which contains the same entries in reverse order. In contrast to the `Cnt` query it never throws a `DelayableException`. It just returns an empty list, if no entries are supplied.

4.2.2.3 Performance Improvements

Unfortunately, the .NET Micro Framework does not support generics (see [49] for generics in .NET and [50] for generics in Java) and therefore type-safe data structures would need to be implemented manually in managed code (using C#). As TinySpaces were implemented with the performance constraints in mind, the decision was made to use the built-in “`ArrayLists`” instead of self implemented *typed* data structures like a list or a hash table. One reason for this is that the self implemented data structures need additional storage and thus raise code-size. Another reason is that they will be

outperformed by the built-in “ArrayLists” in the coming version 4.0 of .NET Micro Framework, as those are going to be re-implemented in native code (using C++). To achieve type safety, type-safe wrapper classes were created, which encapsulate those “ArrayLists”.

The performance will decrease linearly as the number of containers rises, because in order to find one specific container, for example, all containers needed to be iterated through in the worst case. Therefore the current implementation of the IContainerFacade introduces the ExContainerReference (which stands for “Extended-ContainerReference”).

4.2.2.3.1 ExContainerReference

The ExContainerReference inherits from ContainerReference and adds an additional property that stores the container object which it belongs to. Whenever a container is created or looked up, CAPI-1 returns an ExContainerReference containing the container instance, which is, however, not visible to any other layers. If this reference is reused to modify this container, CAPI-1 finds the container in the reference itself and therefore does not have to search for it. However, this obviously only works when accessing the local space as the container instance will not be serialized over the network. As a ContainerReference is passed frequently between the layers to access meta-information of a container – for example CAPI-3 uses it to retrieve the coordinators and CAPI-2 validates the current entry count of bounded containers - during one single read-/write-/take-/destroy- operation and to complete the operation itself, this approach obviously leads to significant performance improvements and better scalability, especially with an increasing number of containers within a space.

4.3 CAPI-2: Transactions

This chapter gives an in depth overview of the contracts as well as the implementation of CAPI-2 of TinySpaces.

4.3.1 Contracts

The following figure shows all necessary contracts.

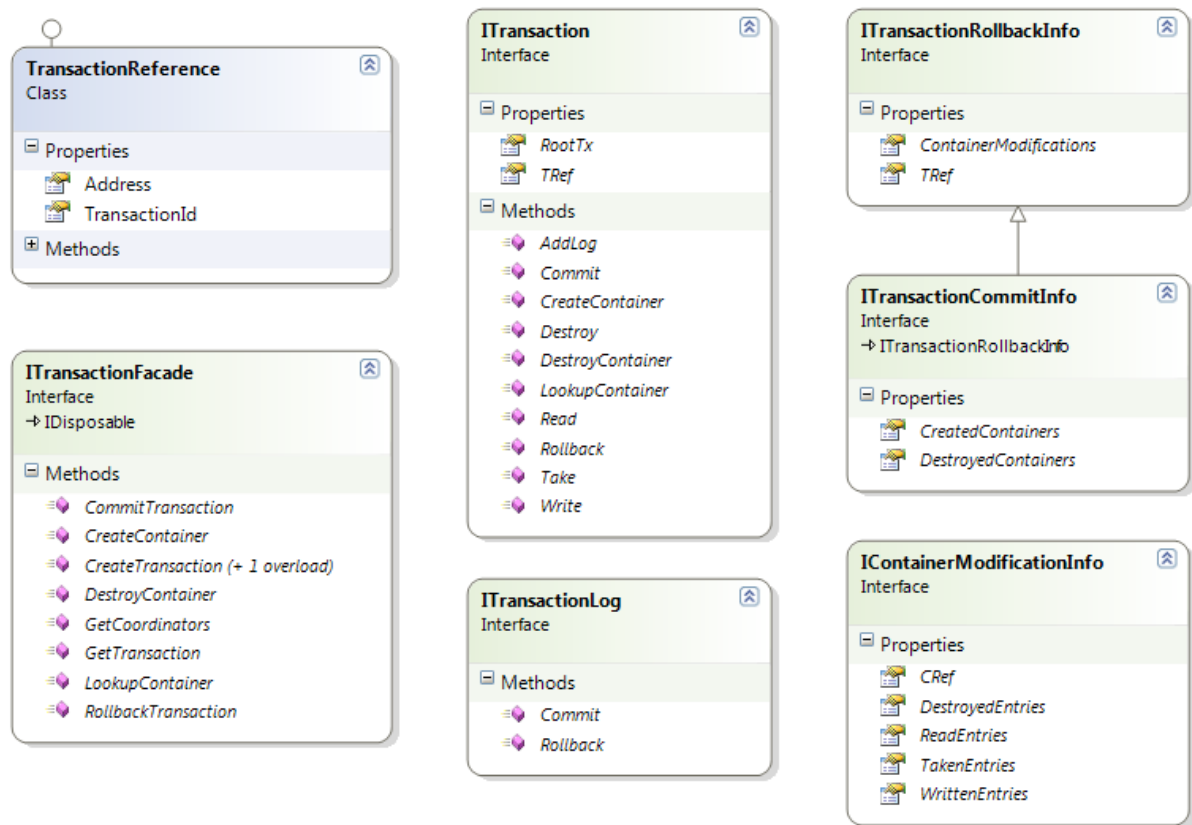


Figure 21- Contracts of CAPI-2

4.3.1.1 TransactionReference

Just like the ContainerReference class, the TransactionReference contains two properties.

TransactionReference		
	Address	Gets/Sets the address of the space this transaction belongs to.
	Id	Gets/Sets the locally unique ID of the transaction.

Table 8 – CAPI-2: TransactionReference

Again the Id-property contains a 32 bit unsigned integer and acts as a locally unique ID for the transaction. Therefore a TransactionReference only gets globally unique if its address is taken into account.

4.3.1.2 *ITransactionFacade*

The ITransactionFacade contains methods to handle transactions in general.









 ITransactionFacade		
	CreateTransaction	Creates a new transaction with a locally unique ID. If a transaction reference is provided, a sub-transaction for the given transaction is created.
	Commit/RollbackTransaction	Takes a TransactionReference, commits/rolls back the corresponding transaction along with all its sub-transactions and removes them. Returns an ITransactionCommitInfo object which will be described later in this chapter. If a sub-transaction is specified, it is furthermore removed from its parent transaction whereby the latter is not committed or rolled back.
	GetTransaction	Searches for a transaction and returns it. If no transaction is found a TransactionNotFoundException is thrown.
	CreateContainer	Creates a new container (whether named or not) using the implementation of IContainerFacade.
	LookupContainer	Looks up a container using the implementation of IContainerFacade.
	DestroyContainer	Destroys a container using the implementation of IContainerFacade.
	GetCoordinators	Takes a ContainerReference as parameter and returns the corresponding ICoordinator objects.



Table 9 - CAPI-2: ITransactionFacade members

As can be seen ITransactionFacade contains methods to create, lookup and destroy containers as well as the ITransaction interface. The reason for this is that the coordination layer (CAPI-3) is not responsible for handling containers in any way. On the other hand the aspect layer (CAPI-4) needs to access this functionality provided by CAPI-2. In order not to propagate ITransaction objects throughout the layers the decision was made to add these method definitions to the ITransactionFacade interface.

Furthermore, the façade contains the method “GetCoordinators” as the coordination layer (CAPI-3) cannot access the container directly if the layered architecture is not to be violated. In another valid approach the coordinators could be stored along with a ContainerReference as identifier within CAPI-3, but that way they would be unnecessarily stored in two places. Besides this would decrease performance as CAPI-3 had to search through its own data structure additionally to find the appropriate coordinators.

4.3.1.3 *ITransaction*

The ITransaction interface defines methods responsible for handling the transaction itself and methods responsible for accessing a container and its contained entries in an ACID way.

 ITransaction		
	RootTx	If it is a sub-transaction the root-transaction is returned. Otherwise the transaction returns itself.











 TRef	Returns the corresponding TransactionReference.
 Commit/Rollback	Commits/Rolls back the transaction along with all sub-transactions. If this is called on a sub-transaction, it is furthermore removed from its parent transaction which is not affected any further.
 AddLog	Adds an ITransactionLog object to the transaction.
 CreateContainer	Takes several parameters (coordination types, maximum size, etc.) and returns a status flag indicating whether the operation has been successful.
 LookupContainer	Takes a name as parameter and returns the ContainerReference of the corresponding container of the local space along with a status flag. If that operation is called from a remote space, the address of the ContainerReference is injected by the CommunicationCore at the runtime layer. (See Error! Reference source not found.)
 DestroyContainer	Takes a ContainerReference as parameter and returns a status flag indicating whether the operation succeeded.
 Write	Takes a ContainerReference and an IEntry as parameters and writes this entry into the corresponding container. Returns a status flag indicating whether the operation has been successful.
 Read	Takes a ContainerReference and IXQuery objects as parameters and returns the list of read entries along with a status flag indicating whether the operation has been successful.
 Take	Takes a ContainerReference and IXQuery objects as parameters and returns the list of taken entries along with a status flag indicating whether the operation has been successful.
 Destroy	Takes a ContainerReference and IXQuery objects as parameters and returns the list of destroyed entries along with a status flag indicating whether the operation has been successful.

Table 10 - CAPI-2: ITransaction members

4.3.1.4 ITransactionLog

The ITransactionLog interface specifies which methods need to be implemented to implement do/undo functionality a transaction can use when it is committed or rolled back.




 ITransactionLog	
 Commit	This method will be called by the corresponding transaction when it commits.
 Rollback	This method will be called by the corresponding transaction when it rolls back.

Table 11 - CAPI-2: ITransactionLog members

4.3.1.5 ITransactionRollbackInfo

Whenever a transaction rolls back, it needs to gather specific information about that action and provide it to the upper layers. For example the runtime layer can use that information to determine which

waiting requests need to be rescheduled. The required information on a rollback is defined by `ITransactionRollbackInfo`.




 ITransactionRollbackInfo		
	TRef	Returns the <code>TransactionReference</code> of the transaction which has been rolled back.
	ContainerModifications	Contains a list of <code>IContainerModificationInfo</code> objects which describe changes made to a specific container which have been undone.

Table 12- CAPI-2: ITransactionRollbackInfo

4.3.1.6 *ITransactionCommitInfo*

This interface inherits from `ITransactionRollbackInfo` and therefore provides the same information and adds additional.




 ITransactionCommitInfo		
	CreatedContainers	Contains the <code>ContainerReference</code> of each container which has been created within the transaction.
	DestroyedContainers	Contains the <code>ContainerReference</code> of each container which has been destroyed within the transaction.

Table 13 - ITransactionCommitInfo

4.3.1.7 *IContainerModificationInfo*

This interface defines information which needs to be collected when a transaction is committed or rolled back.







 IContainerModificationInfo		
	CRef	Returns the <code>ContainerReference</code> of the container which has been modified.
	DestroyedEntries	Contains the entries that have been destroyed within the transaction.
	ReadEntries	Contains the entries that have been read within the transaction.
	TakenEntries	Contains the entries that have been taken within the transaction.
	WrittenEntries	Contains the entries that have been written within the transaction.

Table 14 - CAPI-2: IcontainerModificationInfo

4.3.2 **Implementation**

Although the contracts define exactly how the transaction layer needs to look like on the surface, there are many different ways it could be implemented.

First of all, the transactions could use different **locking strategies**:

- Pessimistic locking

In this strategy the entries and containers which are accessed within a transaction are locked exclusively whereby no other transaction can access them. That way the transaction can *commit* without the risk of running into a conflict with another transaction. A drawback is the risk of deadlocks which can occur when two or more transactions try to acquire locks for the same set of resources in a different sequence concurrently. However, pessimistic locking is an excellent strategy for environments where data is changed frequently by concurrent operations..

- Optimistic locking

If a transaction uses optimistic locking the entries and containers which are modified or read are not locked. This increases concurrency as all other transactions still have access to them. The drawback of this approach is that upon a commit a transaction needs to validate that all modified entities are still in the original state and therefore have not been modified by another transaction. Because of this additional validation process, committing is more costly than it would be using pessimistic locking. That is why this strategy is suited for environments where it is assumed that transactions can most of the time complete without affecting each other. Examples for this locking strategy are modern database systems.

In an XVSM space multiple transactions usually access a container using its predefined coordinators. Furthermore, the sets of entries which are accessed and modified by different user applications typically overlap. (For example when a FIFO-Coordinator is used) Therefore there is obviously a high risk, that two transactions could access the same entries. This led to the decision to use pessimistic locking.

The second decision concerns the **isolation level** that is to be used. The following list describes the common isolation levels from the lowest to the highest according to [51].

- Read Uncommitted

In this level changes made by a transaction are instantaneously visible to all other transactions, even if this transaction has not yet committed them at that time.

- Read Committed

With read committed changes made by a transaction are revealed once if it successfully committed.

- Repeatable Read

While read uncommitted allows transactions to interleave each other, repeatable read keeps them totally isolated from each other although they are executed concurrently. It ensures that during the lifetime of a transaction the same query will always return the same results, even if another transaction has committed changes which would satisfy that query.

- Serializable

This is the highest isolation level. Interleaving transactions are, from a logical view, executed sequentially, thus allowing only one transaction at a time to access the data. This isolation level provides the highest consistency but on the other hand the least concurrency.

As high concurrency is a very important property for a coordination space the isolation level Serializable was never up for discussion as it offers the least amount of concurrency.

Read Uncommitted would provide high concurrency but also the least consistency. Consider for example two transactions *t1* and *t2*. *t1* writes an entry into a container *c1* which in turn is instantly visible to *t2*. Next *t2* reads this entry from *c1*. As we are using pessimistic locking, a read-lock is added to this entry. When *t1* now wants to roll back, it cannot access the entry as *t2* has a read lock on it.

Also Repeatable Read was discarded because of the performance costs – a transaction either would need to take a snapshot of the current state of the accessed containers to be able to always return the same results or use a stricter locking mechanism thus reducing concurrency - as well as the higher implementation effort.

Therefore the TB made the decision to use read committed as isolation level because it suited the application area best, as it allows for high concurrency and an acceptable amount of consistency.

4.3.2.1 Transaction

The Transaction class implements the ITransaction interface and provides access to containers and entries with the isolation level Read Committed and pessimistic locking for CAPI-3.

To store locking information for a container and/or entry, a LockInfo object is created and set in their LockInfo property, which they inherit from the ILockable interface as can be seen in Figure 22- Using ILockable in CAPI-2. This is done when the container is created and/or the entry is written into a container. When the container or entry is destroyed, the LockInfo object is also removed. This LockInfo object is used by CAPI-2 to keep track of which transaction has a lock on the corresponding resource (container or entry).

For this purpose it provides methods to add and remove read-, write-, and delete-locks, which return a Boolean flag indicating whether the lock could be added or removed successfully. It accepts multiple read-locks and a single write- or destroy-lock on an ILockable whereas only one kind of lock can exist at the same time. This means that there can never be a read- and a write-lock on the same ILockable instance, just to give an example. However, a transaction can update its locks. For example a read-lock can be upgraded to a destroy-lock.

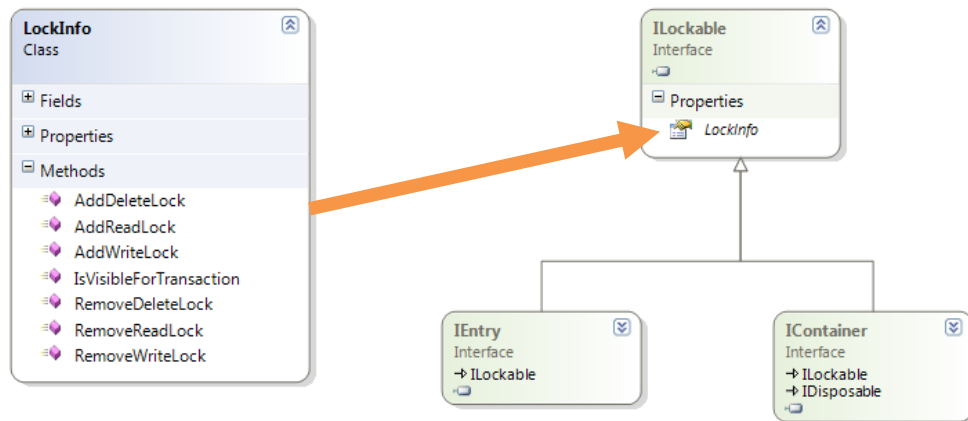


Figure 22- Using ILockable in CAPI-2

To understand how CAPI-2 uses this object, consider a transaction $t1$ which wants to write an entry $e1$ into the container $c1$. To make sure that the $c1$ is not destroyed by another transaction, $t1$ tries to add a read lock to the container by calling “AddReadLock” on the container’s LockInfo object. If it succeeds the transaction uses the entry’s LockInfo to add a write-lock to it and uses the methods provided by $c1$ (IContainer) to write the entry into $c1$. This iterative strategy along with a lot of other rules was conceived by Stefan Craß and can be found in [52] and [18].

Furthermore, the LockInfo is also used to assure that the isolation level of Read Committed is complied with by providing the method “IsVisibleForTransaction”. Before a transaction accesses a container, it uses this method to validate that the container is visible for it. This would for example not be the case if the container was created within another transaction, which has not committed at that time. The same action needs to be taken for entries, but the only way to specify which entries should be returned is to use an IXQuery object. Therefore the transaction layer (CAPI-2) provides its own “read committed query” which is always executed ahead of all other queries coming from the upper layers. This query then filters out all invisible entries. The formal specification of XVSM is fully complied with by that approach.

The following figure (Figure 23) shows an example where entries are successfully read from a container using a single atomic read operation.

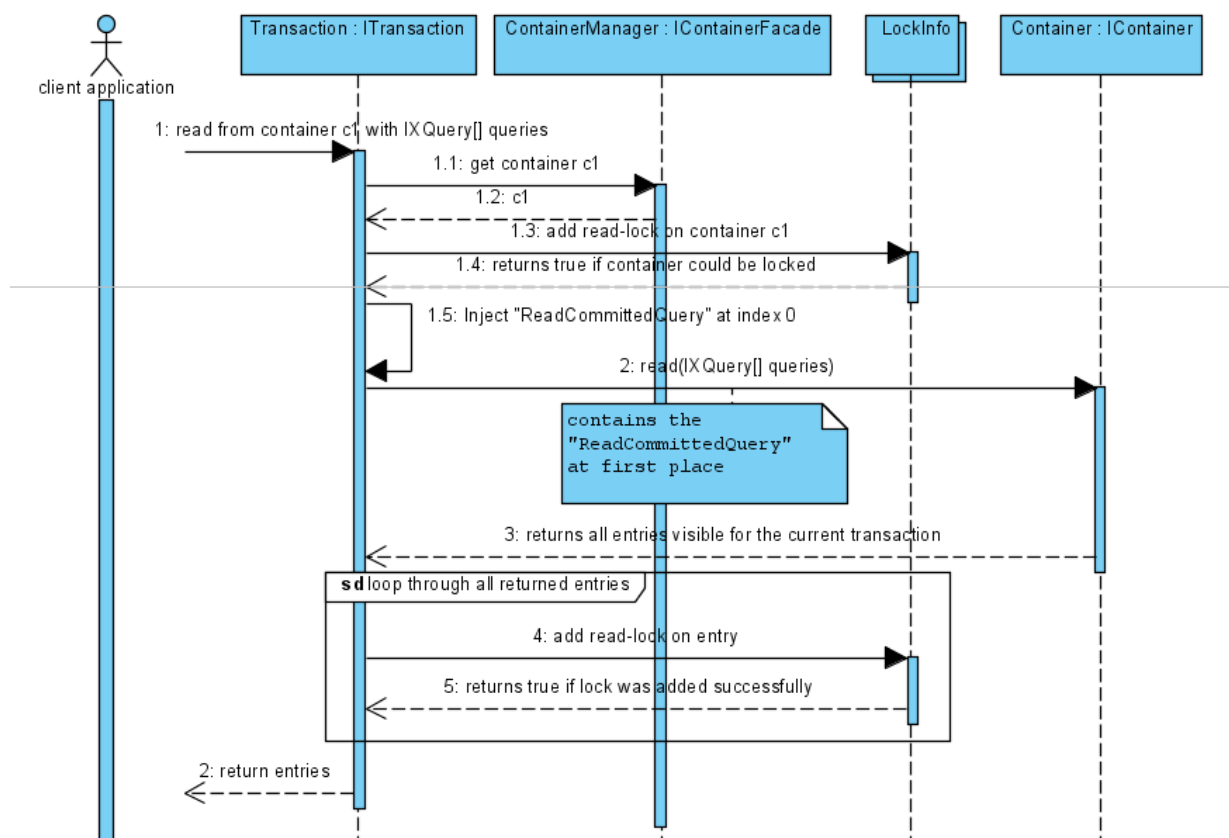


Figure 23 - Reading from a container with a transaction

As can be seen a read-operation is requested by the client application which is eventually received by a transaction object (ITransaction). This transaction first retrieves the container (IContainer object) from the container manager (CAPI-1) and next adds a read-lock to the LockInfo object that is attached to the container. Next the transaction inserts its "ReadCommittedQuery" at the first position of the query-list provided by the upper layers, which filters out any entries that shall not be visible for the current transaction according to the isolation level. Next the transaction provides the list of queries to the container to retrieve the intended set of entries. If any entries are returned it adds a read-lock to each of their attached LockInfo objects. Finally these entries are forwarded to the upper layers and eventually reach the user application.

This sequence of actions needs to be atomic from the client application's point of view. Therefore the transaction has to undo any changes if anything goes wrong. For example if the third entry returned from the container cannot be locked - maybe because another transaction has a delete-lock on it - the transaction must unlock all entries that have already been locked successfully as well as the container itself.

4.3.2.2 Transaction Logs

The last example showed the sequence of activities a transaction has to perform for a single read operation. However, more than one operation (Read, Write, Take, Destroy, Create-/Destroy-/Lookup

Container) can be called within a transaction. Therefore a transaction keeps track of changes made in these operations by storing a dedicated `ITransactionLog` object for each action. These objects encapsulate the information needed to do/undo an action, like adding a read-lock to an entry, and provide methods to commit/rollback their corresponding action. This approach is also used by `XcoSpaces` and conforms to the formal model.

Considering the previous example (Figure 23 - Reading from a container with a transaction), $t1$ would add a log entry for removing the read lock from the container as well as log entries for removing the read lock from each read entry.

When the transaction eventually commits, it simply calls all “Commit” methods of the log entries it has gathered. As pessimistic locking is used, there is no risk that any of the log entries could fail to commit its changes, as it is guaranteed that the transaction has exclusive access to all necessary resources (containers and entries). With optimistic locking on the other hand the transaction would now have to validate its changes.

Although the use of pessimistic locking eases the process of committing there is one remaining important point: The logs need to be classified into two distinct groups, which also specified in the formal model [18].

- Lock-Log Entries (formally logs of type “LOCK-INSERTED”)

These are the logs which are used to remove any kind of lock from an entity or container.

- Normal Log Entries (formally logs of type “PROP-INSERTED” and/or “PROP-DELETED”)

This group contains all remaining types of log entries like one for removing a written entry if a transaction is rolled back, just to give an example.

To understand the necessity for this distinction, consider the following example. Within a transaction $t1$ an entry $e1$ is taken from a container $c1$. Therefore a lock-log for removing the destroy lock from $e1$ on commit is stored as well as a take-log which will finally remove $e1$ on commit. When $t1$ now commits it iterates through its gathered logs and calls their “Commit” method. Therefore $t1$ would remove the destroy-lock from $e1$ before $e1$ is physically removed from the container. During these two actions another transaction $t2$ could add a lock on $e1$ and thus would expect to have exclusive access to it. On the other hand $t1$ would also expect to have exclusive access and removes $e1$. This would lead to an inconsistent state. The most important part of this example, which is the removal of the delete-lock by $t1$ followed by the insertion of a read-lock by $t2$, is depicted in the following figure. (Figure 24)

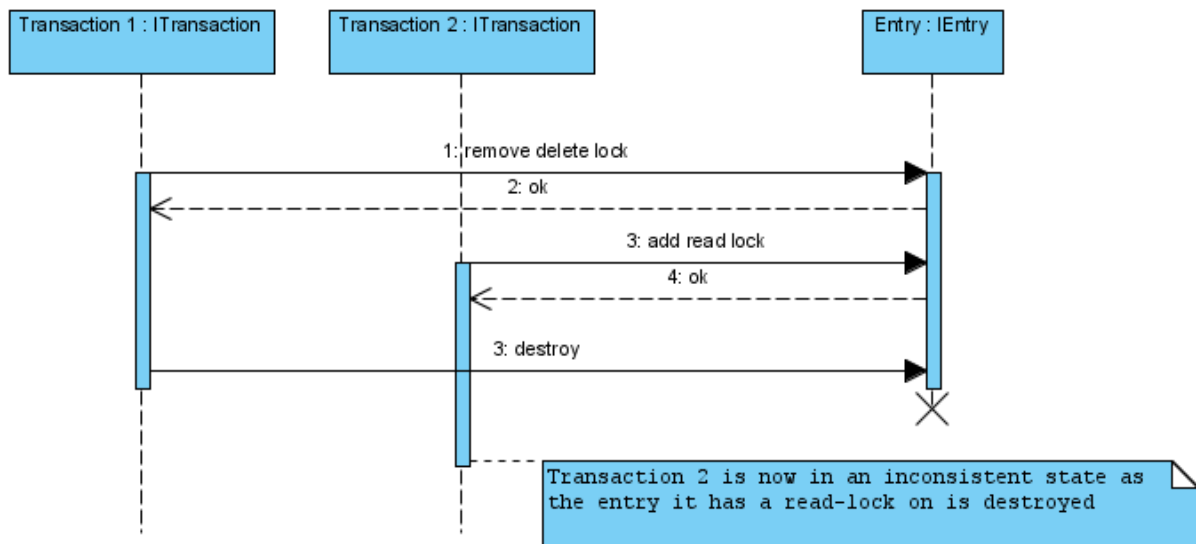


Figure 24 - Transaction in inconsistent state

To avoid that a transaction keeps track of those two types of logs separately and when it eventually commits, it executes the normal logs first and the lock-logs second.

4.3.2.3 Performance Considerations

The last part of the chapter about CAPI-1 describes considerations made to improve runtime performance and reduce code-size or in general: To better meet the requirements of embedded devices.

4.3.2.3.1 Gathering Meta-Information of a Commit or Rollback

As already described in previous chapters a transaction needs to provide meta-information about a commit or rollback to the higher layers as for example which containers were created or destroyed during the transaction or which entries were read/taken from or written to a container. This information, however, is spread across all its transaction log entries. Furthermore, not all logs contain information which is required by the upper layers. For example, information of coordinators' log entries, which are used to do/undo the insertion or deletion of accountant information, are ignored.

The transaction logs which need to provide information for a commit or rollback need to implement the following two interfaces:



Figure 25 – Commit/Rollback Information Provider Interfaces

The `ITransactionCommitInfoProvider` defines the method “ProvideInfos” which takes an `ITransactionCommitInfo` object as parameter. It is assumed that the transaction log, which implements

this interface, adds all viable information to this object. For example a take-log adds the corresponding entry to the “TakenEntries” collection.

The `ITransactionRollbackInfoProvider` works similar to this but takes an `ITransactionRollbackInfo` object as parameter.

This approach reduces code-size and raises runtime performance since no conditional statements are needed to differentiate between the log entries, which would be the case if the transaction object had to gather the information itself. In that case it would need to use conditional statements to determine the nature of a particular log entry and the information that can be extracted from it, which would result in many lines of code. Using the current approach, the log entry only has to be casted to an interface (`ITransactionRollbackInfoProvider` or `ITransactionCommitInfoProvider`) and the “ProvideInfos” method needs to be invoked. Therefore no expressions need to be evaluated (for conditional statements) as the log entries know exactly which information they have to provide.

4.3.2.3.2 General Use of Transactions

To guarantee transactional integrity any operations affecting containers and entries always need to run within a transaction. However a client application might also call an operation without explicitly specifying a transaction which will be explained in future chapters about the runtime and the API. In that case a transaction needs to be created and committed implicitly for that single operation. This behavior is intrinsic and therefore defined in the formal model and also implemented by `XcoSpaces` and `MozartSpaces`.

Unfortunately when implicit transactions are used for a request, three operations have to be performed instead of just one: First the runtime layer has to create a transaction, then the requested operation (e.g.: a write operation) is performed and as the third and final step the transaction has to be committed or rolled back by the runtime layer. This obviously leads to higher computational costs. Moreover the runtime does not obtain a direct reference to the transaction object, because this would violate the layered architecture. It only gets the `TransactionReference` identifying it. Consequently that transaction object needs to be searched by CAPI-2 twice to perform these three operations, namely when the actual operation is performed using that transaction, and when the transaction is eventually committed implicitly by the runtime layer.

This may result in poor performance, especially if many transactions exist at the same time. Therefore the transaction layer provides an extended version of the `TransactionReference`: The `ExTransactionReference` (which stands for “Extended `TransactionReference`”). As the `ExContainerReference`, which has been introduced in 4.2.2.3.1, this object contains a direct reference to the `ITransaction` object and allows CAPI-2 for skipping the search.

To understand how the use of `ExTransactionReference` improves runtime performance of `TinySpaces`, consider a read operation executed without any transaction specified by a user application. The runtime detects this and implicitly makes a call to create a new transaction. Next the transaction layer processes this call and returns an `ExTransactionReference` which already contains the direct reference to the new transaction *t1*.

Next the runtime executes the read-operation within *t1*. Therefore, the corresponding read-operation of CAPI-2 is called with the *ExTransactionReference* object as argument. The transaction layer, however, does not need to search for that instance as it is already provided by the reference itself and therefore can just return it.

Finally the runtime commits the transaction. For that reason it calls the commit operation of CAPI-2 with the *ExTransactionReference* object as argument. Again the transaction layer does not have to search for the instance of the transaction anymore and just returns the one provided by the *ExTransactionReference*.

The following figure (Figure 26) shows this process. Please note that the layers between the runtime- and the transaction layer have been omitted for brevity.

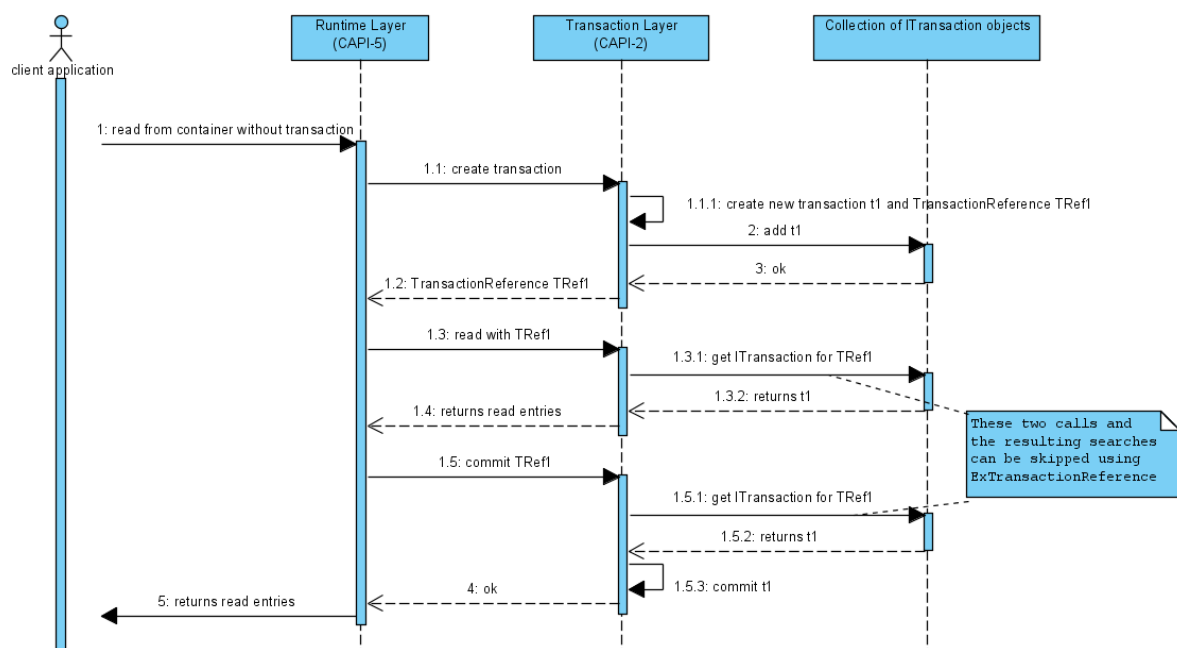


Figure 26 - Implicit transactions and performance

It is obvious that this approach greatly improves performance in case of implicit transactions, but still conforms to the layered architecture of XVSM.

4.4 CAPI-3: Coordination

This chapter gives an in depth overview of the contracts as well as the implementation of CAPI-3 of TinySpaces.

4.4.1 Contracts

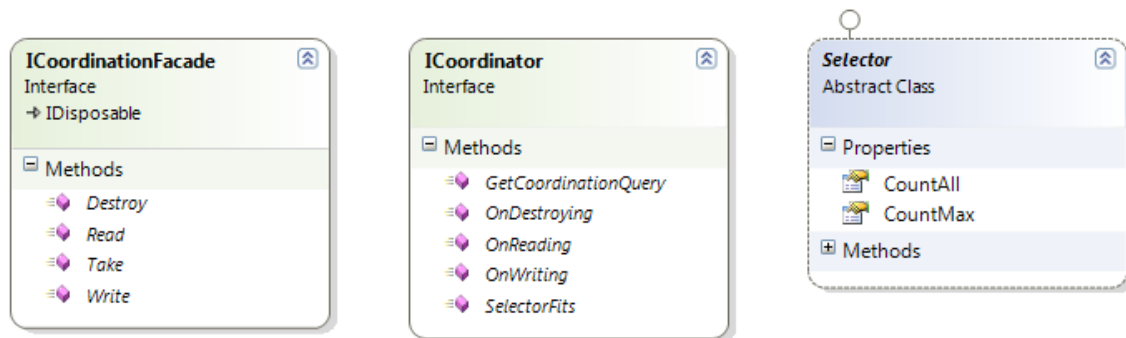


Figure 27 - CAPI-3 Contracts

4.4.1.1 ICoordinationFacade

The **ICoordinationFacade** interface provides the methods which are used by the higher layers (especially CAPI-4) to access this layer.

ICoordinationFacade		
	Take/Destroy	Takes a ContainerReference along with a list of IQuery objects and returns the successfully destroyed entries and a status flag indicating whether the operation has been successful.
	Read	Takes a ContainerReference along with a list of IQuery objects and returns the successfully read entries together with a status flag indicating whether the operation has been successful.
	Write	Takes a ContainerReference along with an entry (IEntry) as parameter and returns a status flag indicating whether the operation has been successful.

Table 15 - CAPI-3: ICoordinationFacade

As can be seen methods for creating, looking up and destroying containers are missing. This is because the scope of this layer is restricted to coordinating access to the contents (the entries) of containers. Therefore, these three operations are not really needed within CAPI-3. The drawback of this approach is that the clean layered architecture of XVSM is violated because CAPI-4 directly accesses CAPI-2 for modifying containers as can be seen in Figure 28 - CAPI-3: Violating the layered Architecture.

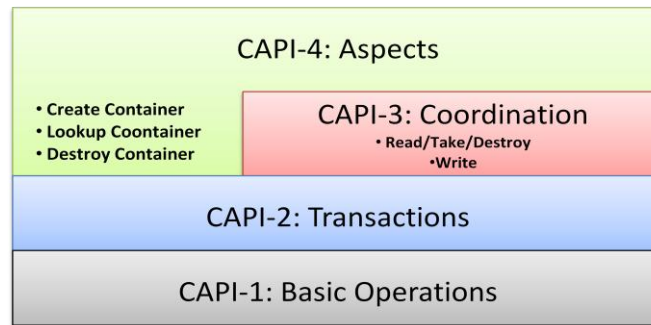


Figure 28 - CAPI-3: Violating the layered Architecture

However it also brings two advantages: The first is that not complying with the layered architecture leads to a slightly better performance as a call from CAPI-4 does not need to be routed through CAPI-3.

The second and major advantage is that extensibility is enhanced as a developer implementing a new coordination layer does not need to take care of modifying containers, but can concentrate on the real scope of this layer.

4.4.1.2 ICoordinator

The ICoordinator interface plays an important role as a major benefit of XVSM is the ability to extend coordination functionality by implementing custom coordinators. Therefore, it is assumed that this interface specifies a central extensibility point which will be used frequently and even by developers who are not specialists for XVSM. For example, a scenario where a custom coordinator needed to be implemented is described in [17]. This leads to the following definition of ICoordinator:




ICoordinator		
	SelectorFits	Takes a Selector and returns true if the selector can be used by the current coordinator. (e.g.: A Key-Coordinator will return true only if a Key-Selector is specified)
	GetCoordinationQuery	Takes a selector and returns an IXQuery object that expresses which entries the coordinator wants to get based on the information of the selector.
	OnWriting/OnReading/OnDestroying	Take an entry and return an array of log entries (implementing ITransactionLog) thus giving the coordinator the chance to perform certain actions in case the current transaction is committed or rolled back. For example, when an entry was written to a container using a transaction and that transaction is rolled back, the corresponding accountant information needs to be removed from the coordinator.

Table 16 - CAPI-3: ICoordinator members

As can be seen, the ICoordinator is absolutely passive, meaning that the methods defined are only used by the coordination layer to inform a coordinator about any actions made, thus giving it a chance to *react* but not to *act*.

This approach has an advantage and a disadvantage: On the one hand a developer implementing a coordinator has not much freedom in developing it. For example there is no possibility for a coordinator to use a transaction for accessing its accountant information, as the OnWriting/OnReading/OnDestroying methods do not take a transaction instance as parameter.

This, on the other hand, greatly reduces the risk of an erroneous implementation, as most functionality is already built into the coordination layer itself and does not have to be re-implemented by the developer.

4.4.1.3 Selector

The Selector contract plays another important role as it is used to provide additional information for coordinators when entries are written to and read/taken from the container.



Selector		
	CountAll	Specifies whether all entries of a container, which are visible for the current transaction, should be returned. Entries with a write-lock attached are filtered out. However, if an entry has a destroy-lock assigned, the operation (read/take/destroy) will result in LOCKED.
	CountMax	Specifies whether the maximum count of available entries should be returned. The difference to CountAll is that entries with a destroy-lock assigned are simply filtered out and the operation (read/take/destroy) succeeds.

Table 17 - CAPI-3: Selector

The interface only specifies two Boolean properties which are used to indicate how many entries should be read and thus need to be implemented by the coordination query of each coordinator.

In XcoSpaces a Selector also contains a “Count” property specifying the expected number of entries returned. This integer value is further used to indicate “CountMax” (which is called “CountAll” in [7]) by assigning the value of “-1”. The semantic of “CountAll”, as it is defined in the formal model, is not implemented in XcoSpaces yet.

This approach was not used for TinySpaces as a 32 bit signed integer is needed to store the negative values for “CountMax” and “CountAll”. This would be a waste of memory and bandwidth as a 16 bit unsigned integer along with two additional bits for “CountMax” and “CountAll” (therefore 18 bit) would be absolutely sufficient.

A selector is always tightly coupled to a coordinator and vice versa. When a new coordinator is implemented, a specialized selector for it also has to be developed to provide additional query information alongside the two mentioned properties. For example a FIFO-Selector contains a number indicating the exact amount of entries required whereas a Key-Selector contains keys identifying the required entries.

As already mentioned, selectors play an important role. On the one hand, they provide the information needed by the corresponding coordinator to create a query specifying the entries, which should be read, taken or destroyed. On the other hand they also provide information about how entries need to

be written. For this reason the IEntry interface (see 4.2.1.4) contains a property holding a list of selectors, which are called “write selectors” in that case.

Consider for example a Key-Coordinator which identifies an entry by its corresponding key. When an entry is written to a container coordinated by a Key-Coordinator, a key which shall be used to identify that entry needs to be propagated to the coordinator. Therefore selectors (or “write selectors” according to the formal model) can be attached to an entry. When the coordination layer eventually calls the “OnWriting” method of the Key-Coordinator, the coordinator searches through the attached write selectors of that entry for a Key-Selector, which specifies the key that shall be used as an identifier for that entry. However, it could happen that a required selector is missing thus causing an error which needs to be handled by the coordination layer. This will be described in 4.4.2.2.

4.4.1.3.1 The Relation of Selector and IXQuery

CAP1-2 and CAP1-1 have no knowledge of selectors and coordinators according to the layered architecture specified in the formal model. Therefore, the coordination layer has to generate queries to tell CAP1-1 and CAP1-2 which entries shall be read, taken or destroyed. To achieve this, the selectors given for the current operation (read/take/destroy) are forwarded to their corresponding coordinators. These use the information provided by the selectors to generate the appropriate queries (IXQuery objects) which are gathered by the coordination layer and forwarded to CAP1-1 and CAP1-2.

Moreover, it shall be possible to specify a mixture of queries and selectors for read-, take-, and destroy-operations (IXQuery objects) at the API layer (for example a Cnt query as described in 4.2.2.2.1) to achieve the maximum amount of expressiveness. A client application may make a read-operation specifying a Cnt query along with a FIFO- and a Key-Selector in any order, just to give an example.

In the formal model and the current implementation of XcoSpaces, queries are represented by selectors at API level and generated by a Query-Coordinator in CAP1-3. As that coordinator does not gather any accountant information about the entries of a container, it cannot provide any useful information for a query. Therefore, such a coordinator was omitted in the implementation of TinySpaces and the queries are directly created by the user application and/or CAP1-3 and CAP1-2.

However, the Selector and IXQuery contracts need to be related to each other and in turn derive from a common contract (IQuery), which is shown in the following figure.

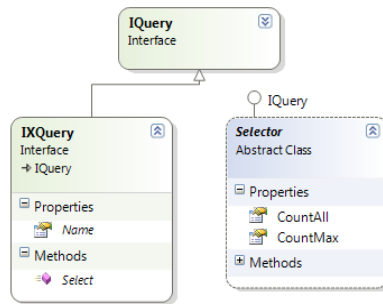


Figure 29 - CAPI-3: Relation of IXQuery and Selector

There is an important reason for this solution:

A selector cannot derive from **IXQuery** directly as it is not a query itself, but can be used by a coordinator to create a query. If a selector would implement this interface, it would mean that a selector is some sort of specialized **IXQuery** and thus it would be assumed that the “Select” method could be called to get a viable list of results, which is not the case. A selector (in the context of a read-, take-, or destroy-operation) is only a piece of context information, which is completed by the context information a coordinator gathers about the entries of a container to in turn create a query.

In contrast to this, a simple query object like a **Cnt** query can only get context information from the client application and thus does not have the benefit of having access to the accountant information kept by coordinators of a container.

To get a better understanding of this consider a **FIFO-Selector**. As a **Cnt** query this selector can store a number specifying the amount of entries required. However the query generated by the **FIFO-Coordinator** can return different results than the **Cnt** query:

Consider that a read operation is called with a **Key-Selector** containing the keys $k1$ and $k2$ and a **Cnt** query with a count of 1. First the **Key-Coordinator** creates a query which selects all entries identified by the keys $k1$ and $k2$ as specified by the selector. These entries are then handed over to the **Cnt** query which returns the first of the two entries ($k1$) in the order they are provided.

If this **Cnt** query was replaced by a **FIFO-Selector** with the same value for “count”, a query would be generated which could return a different entry. This is because the resulting query would not return the first of the two entries in the order they were provided, but the very entry that has been written to the container first. If the entry $e2$ identified by $k2$ had been written to the container before the one identified by $k1$, $e2$ would be returned.

The following figure shows the difference.

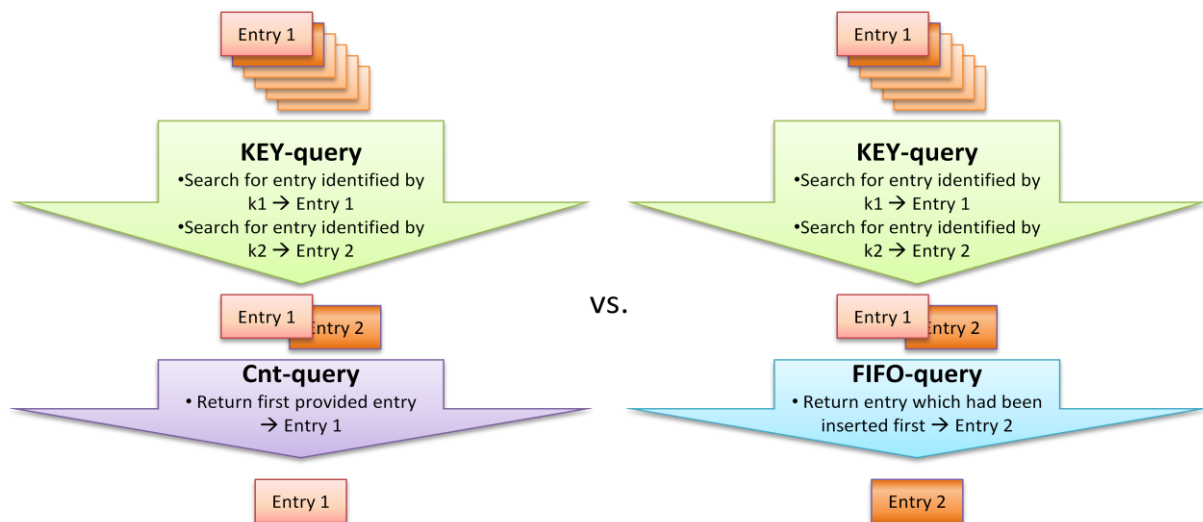


Figure 30 - FIFO-query versus Cnt-query

Moreover, a selector (or “write selector”) is used to carry additional information for a coordinator, when an entry is written to a container. Remember that a Key-Coordinator requires the written entry to contain a Key-Selector specifying a key which shall be used to identify that entry. Therefore an IXQuery cannot be seen as a specialized form of Selector as this would make it possible to specify a Cnt query within an entry although it does not keep any viable information for any coordinator.

4.4.2 Implementation

This chapter provides an overview of the current implementation of CAPI-3 in TinySpaces.

4.4.2.1 Provided Coordinators

The current implementations of XSVM (MozartSpaces and XcoSpaces) provide a wide variety of coordinators out of the box. However, as TinySpaces targets embedded devices, its code-size has to be minimal. For that reason it had to be decided which ones of the coordinators defined in the formal model of XSVM are absolutely necessary and therefore have to be built-in. The following list provides a brief overview over the existing coordinators to offer the reader enough knowledge to understand that decision. For more detailed information about those coordinators we refer to [18] and [10].

FIFO-Coordinator (First-In First-Out): This coordinator has already been introduced in previous chapters. It keeps track of the time the entries were written to a container and returns the entries in that order. Therefore it acts like a kind of queue.

LIFO-Coordinator (Last-In First-Out): The LIFO-Coordinator is the counterpart of the FIFO-Coordinator as it also keeps track of the insertion time of the entries, but first returns those entries that have been written most recently. Therefore it resembles a stack.

Key-Coordinator: As already mentioned in previous chapters, this coordinator uses unique keys to identify the coordinated entries. Therefore it resembles a dictionary.

LABEL-Coordinator: This coordinator works similar to the Key-Coordinator as it uses keys to identify entries within a container. However unlike the Key-Coordinator it does not require these keys to be

unique wherefore they are named “labels”. For this reason several equal labels can exist and therefore multiple entries could be returned even if only one label was specified in a selector.

LINDA-Coordinator: The LINDA-Coordinator provides the coordination pattern implemented in JavaSpaces. It uses template matching to specify which entries shall be returned.

List-Coordinator (or Vector-Coordinator): A List-Coordinator behaves similar to a linked list. It uses index values, which have to be unique of course, to identify the entries of a container. The difference to a Key-Coordinator is that a key assigned to an entry never changes, whereas an index value assigned to an entry can change. For example, when an entry at index 2 is removed the successive indexes (3 to maximum index) are adjusted. (3 changed to 2, 4 changed to 3, etc.)

RANDOM-Coordinator: This is another coordinator known from previous implementations. Rather than providing the ability to directly select entries from a container, this coordinator returns entries in a random manner.

The decision was made that only two of these coordination types are absolutely necessary for TinySpaces.

The **FIFO-Coordinator** was chosen, as there is a variety of scenarios where a producer consumer pattern is required. Consider for example a temperature sensor which periodically writes its measurement data into a container. Using a FIFO-Coordinator a consumer receives those entries in the correct order. Furthermore this coordination type is required for the implementation of notifications which will be described in 4.5.3.

The second required coordinator is the **Key-Coordinator** as applications frequently have the need to store entries identified by a unique key. A good example for this would be a meta-container which contains entries that are identified by well known keys or lookup containers. Another scenario where this coordination type would be helpful is a space with containers which form a hierarchy. For this, a Key-Container may store ContainerReferences pointing at the sub-containers of that hierarchy.

This could also have been achieved using a LABEL-Coordinator, however while it is simple to replace this kind of coordinator with a Key-Coordinator by adding a list of values to an entry instead of just one, it is impossible the other way around.

Also the LINDA-Coordinator was discarded as template matching can cost a lot of performance and is therefore not that applicable for resource-constrained devices.

4.4.2.2 *The CoordinationFacade - Handling Multiple Coordinators*

The coordination layer needs to handle read-, write-, and take-operations which have multiple selectors assigned and could also be mixed with simple IXQuery objects. Take for example a read operation which specifies a FIFO-Selector followed by a Cnt query and a Key-Selector.

Therefore it first needs to get the coordinators assigned to the destination container using CAPI-2. Then the IQuery objects (consisting of IXQuery- and Selector objects) provided by the read-operation

are searched for selectors, which are in turn handed over to their corresponding coordinators. The coordinators use these selectors to create a query (IXQuery object). As already denoted, this needs to be done as the lower layers (CAPI-2 and CAPI-1) have no knowledge of selectors. Next, the created (coordination) queries are collected and assembled to a list along with the (normal) queries that have been handed over by the upper layer of the operation, keeping the original order. Then that list is handed over to the transaction layer to perform the requested operation (Read, Write or Take). This process is shown in Figure 31.

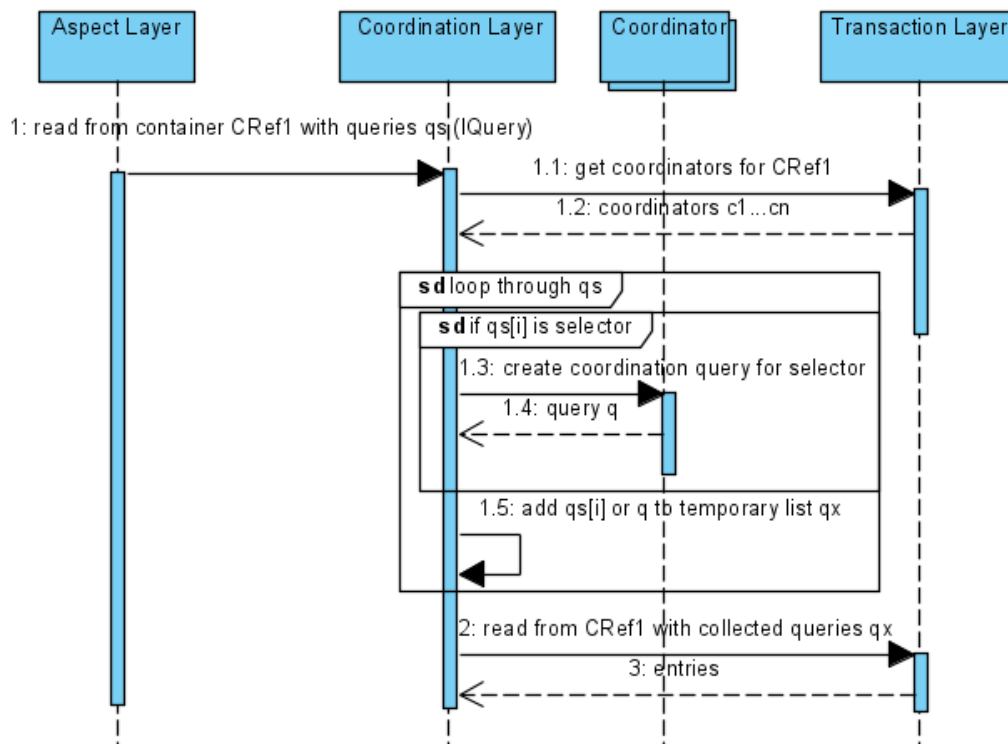


Figure 31 - CAPI-3: Creating queries

Additionally, as a container can have multiple coordinators assigned, the coordination layer needs to synchronize them whenever an entry is added or removed.

Consider for example that a container has a Key- and a FIFO-Coordinator assigned. When an entry is written to the container, the coordination layer needs to make sure that both coordinators are informed about that event so they can keep their accountant data up-to-date.

Unfortunately a lot of errors can occur during this step. Consider that a write-operation is performed on a container which has a Key-Coordinator assigned. For that reason a write selector specifying a key has to be attached to that entry. If this is not the case, the Key-Coordinator cannot proceed and will throw an error. Furthermore, a custom coordinator could be erroneous and throw an exception.

As multiple coordinators can be assigned to a container, it is possible that an error does not happen with the first coordinator, but one of its successors. Therefore the coordinators, which have already

been informed successfully, are in an inconsistent state as the operation is not valid anymore and changes made to their internal accountant information need to be rolled back.

To be able to handle this incidence, the coordination layer needs to make use of transactions for each single operation. However, it is important that a user transaction handed over by the runtime is not directly used for this purpose, because many other operations may have already been performed with it. Therefore, if an error occurs within CAPI-3 and hence the transaction is rolled back, all other valid changes made within this transaction are lost too.

To solve this, the coordination layer creates a new transaction for each read-, write-, take-, and destroy-operation and adds it as sub-transaction to the user transaction. This way all changes made by one single operation are treated separately by the corresponding sub-transaction and therefore can be easily undone without affecting the parent transaction. When the parent transaction eventually commits or rolls back, it does the same with all its sub-transactions. This has been defined in [52 S. 3]. This process is shown in Figure 32.

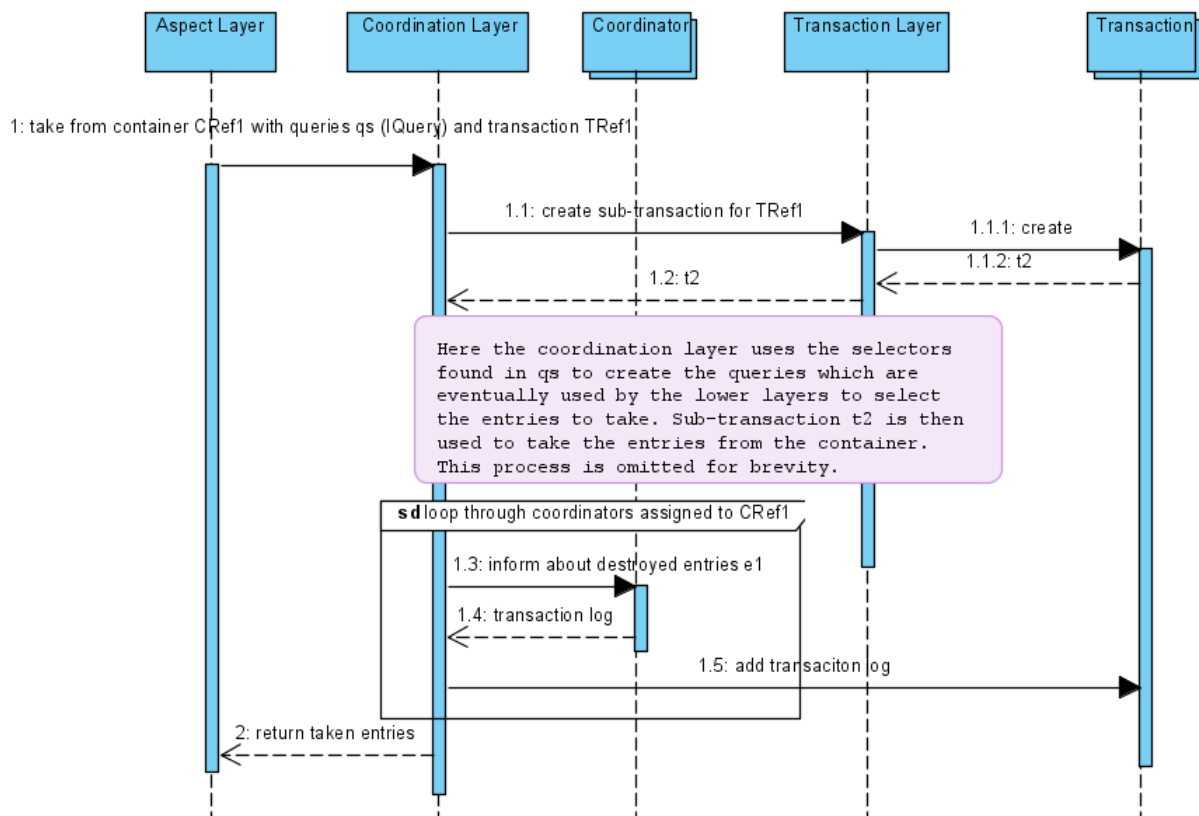


Figure 32 - CAPI-3: Synchronization of Coordinators

4.4.2.3 Isolation Level of CAPI-3

As already mentioned, the coordinators of the current CAPI-3 implementation of TinySpaces do not use transactions to add and/or remove accountant information although this is specified in the formal model. This violates the isolation level of Read Committed, because a coordinator has access to all

accountant information even if it was added by a transaction which has not committed yet. Therefore, the isolation level used for accountant information is Read Uncommitted and not Read Committed.

As using transactions to modify a coordinator's accountant information would cost additional performance, this approach was discarded for TinySpaces as it targets resource-constrained devices. The major drawback of the approach used in TinySpaces is, that accountant information cannot be accessed by a user application using the XVSM API. This is not acceptable for the enterprise versions of XVSM. However the current implementation of XcoSpaces follows the same approach and needs to be adapted to comply with the formal model.

The usage of isolation level Read Uncommitted brings another benefit for TinySpaces: Consider two transactions $t1$ and $t2$ trying to write entries for the same key k into a container coordinated by a Key-Coordinator. If those keys are added to the accountant information using an isolation level of Read Committed the following happens: The coordinator uses $t1$ to add the key k to its accountant information which is therefore still invisible to other transactions. For this reason $t2$ also succeeds to add the key k . When those transactions eventually commit, both keys become visible which would cause the second committing transaction to fail during commit, causing it to be in an inconsistent state. For that reason the formal model defines, that a Key-Coordinator needs to prevent concurrent write access. [18 p. 48] However as the isolation level is Read Uncommitted in TinySpaces, the coordinator instantly detects that this key has already been added and throws a DelayableException causing the sub-transaction to be rolled back and the operation to be delayed.

The approach used in TinySpaces can furthermore not lead to a violation of the lower layers isolation levels. For example it cannot cause entries to be returned which still have to be invisible for the transaction which encloses the current operation (read/take/destroy) as they were written by a different one which has not committed at that time. The reason for this is that CAPI-2 filters out invisible entries *before* the queries provided by CAPI-3 are executed: Consider for example two transactions $t1$ and $t2$. $t1$ wrote an entry $e1$ with the key $k1$ into a container and has not committed yet. Therefore the entry is not visible at this time as the isolation level for CAPI-1 and CAPI-2 is Read Committed. In contrast to that the corresponding key *is* visible for any other transaction as the isolation level of accountant information of CAPI-3 is Read *Un*committed. If $t2$ now wants to read the entry with key $k1$, the coordination layer uses the Key-Coordinator to generate the necessary query and adds it to a list which is handed over to CAPI-2. That in turn inserts its "read committed query", which is used to filter out any invisible entries (according to Read Committed) at index 0 of the list provided by CAPI-3 and hands the queries over to CAPI-1. Accordingly, when CAPI-1 executes those queries, the entry $e1$ is filtered out by the "read committed query" of CAPI-2 and *not* provided for the coordination query of CAPI-3. For that reason the coordination query throws a DelayableException and the read-operation will be rescheduled by the runtime. As a result $t2$ fails to retrieve the entry at that time, which satisfies the isolation level Read Committed.

The only case where this approach could cause problems is when the client application read all keys from the Key-Coordinator to verify that a specific key exists, in order to read that entry only if this is the case. Using Read Uncommitted a key could be returned which still had to be invisible. Therefore the

client application could erroneously assume that an entry with this key can be read and make a read-operation. If the transaction which added that entry along with the key would be rolled back, this read-operation eventually times out. However if that timeout was infinite, the client application could get stuck forever. (Timeout handling will be described in 4.6.3) Fortunately that scenario can be realized in TinySpaces using a read operation with “timeout 0”, which means that it will fail immediately, if the required key does not exist. For more information about “timeout 0” semantics, we refer to [18].

4.5 CAPI-4: Aspects

SoC (separation of concerns) is an intrinsic concept of object-oriented programming [53]. It demands that software is separated into distinct modules containing functionally linked building blocks. The architecture of XVSM complies with SoC as it is separated into several layers. For example all functionality concerning transactions in XVSM is encapsulated in CAPI-2.

An aspect however is a part which cross-cuts the core concerns of a program and thus violates its separation of concerns. Typically this is a feature which does not belong to the core functionality of the corresponding software system, like for example logging, persistence, security, and so on.

The task of the aspect layer (CAPI-4) is to provide and manage well defined points in TinySpaces where aspects can be added. This allows for adding functionality like persistence and security without modifying the existing layers.

This chapter will give an overview of how this is achieved by introducing the contracts of CAPI-4 and their current implementation in TinySpaces.

4.5.1 Contracts

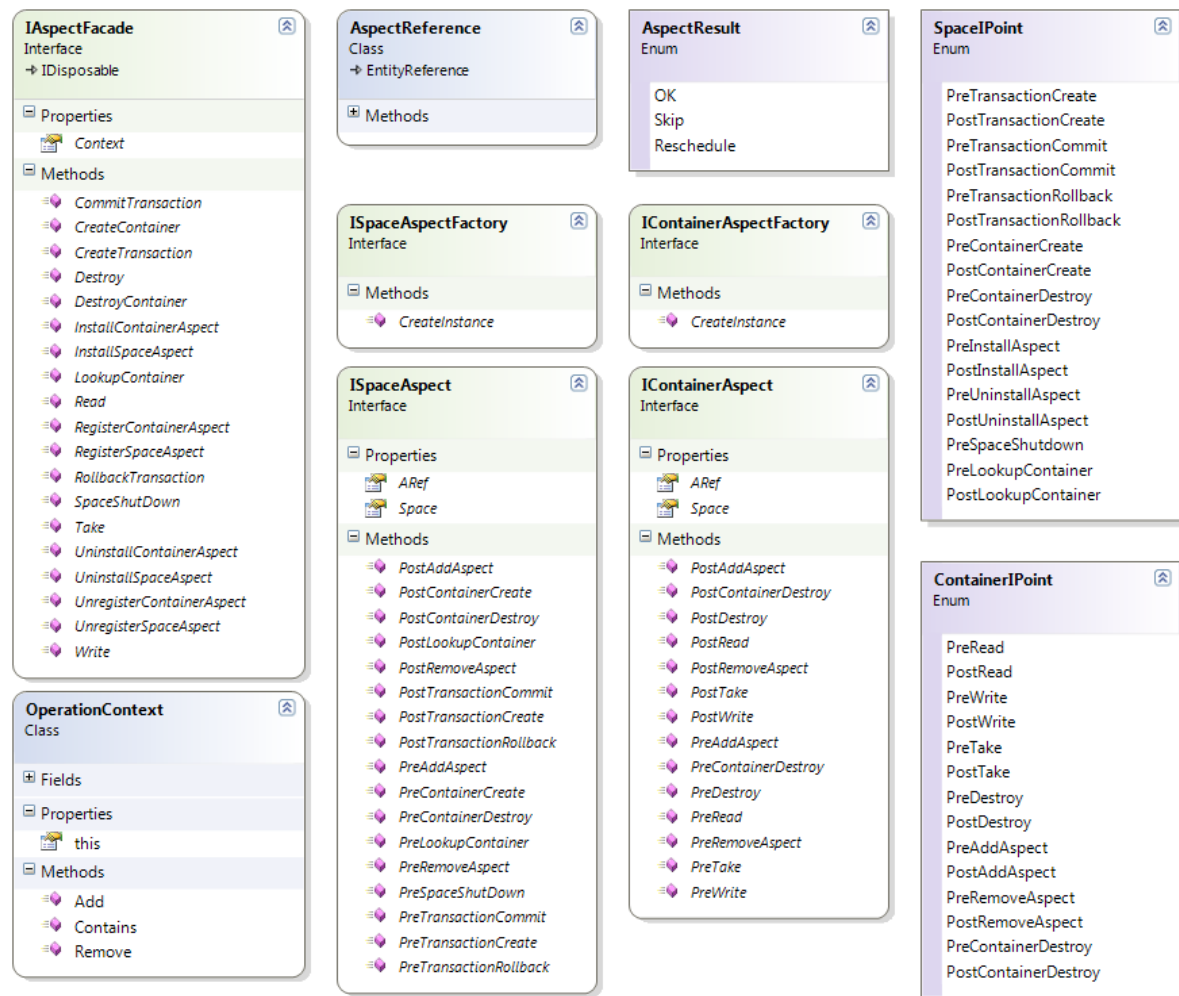


Figure 33 - CAPI-4: Contracts

As most of the contracts for aspects of XcoSpaces, the .NET reference implementation of XVSM, comply with the new layered architecture of XVSM, these contracts have been borrowed and will thus not be described in detail in this chapter. For further information about those we refer to [7].

4.5.1.1 AspectReference

The AspectReference is similar to the ContainerReference and the TransactionReference as it provides a space address and the local ID of the corresponding aspect to identify it.



AspectReference	
 Address	Gets/Sets the address of the space this aspect is located in to.
 Id	Gets/Sets the locally unique ID of the transaction.

Table 18- CAPI-4: AspectReference members

4.5.1.2 *IAspectFacade*

This is the facade used by the runtime layer (CAPI-5) to access the entire functionality of the lower layers. Therefore, the implementation of the façade can use these methods as insertion points for its pre- and post-aspects.











 IAspectFacade		
	OperationContext	Contains an OperationContext object storing key-value pairs which can be used by the aspects to cache state information.
	Create-/Lookup-/DestroyContainer	Provide access to CAPI-1 operations.
	Create-/Commit-/RollbackTransaction	Provide access to CAPI-2 operations.
	Read/Write/Take/Destroy	Provide access to CAPI-3 operations.
	SpaceShutDown	Is called before the local space is shut down.
	RegisterSpaceAspect/UnregisterSpaceAspect	Takes an ISpaceAspectFactory object along with a name identifying the type of aspect the factory creates.
	InstallSpaceAspect/UninstallSpaceAspect	Takes a name identifying the type of the aspect and returns an AspectReference identifying the created instance.
	RegisterContainerAspect/UnregisterContainerAspect	Takes an IContainerAspectFactory object along with a name identifying the type of aspect the factory creates.
	InstallContainerAspect/UninstallContainerAspect	Takes a name identifying the type of the aspect and returns an AspectReference identifying the created instance.

Table 19 - CAPI-4: IAspectFacade members

One method which seems to be misplaced in this contract is “SpaceShutDown”. This is because the aspect layer cannot shutdown the space, as it would need to access the runtime layer. However that is not allowed as this is a higher layer (CAPI-5). On the other hand an insertion point is needed which is called before the space shuts down. For example a log-aspect could need to close the log file as soon as the space shuts down. Additionally “SpaceShutDown” is the only insertion point which does not support post-aspects as they would obviously never be called as soon as the space is not running anymore. Shutdown behavior will be further described in 0.

This contract also provides two new groups of methods which are used to manage aspects. One group handles so called space aspects whereas the other handles container aspects. The difference between these types of aspects is that space aspects are added to insertion points at space level. Container aspects, as the name implies, are added to single containers. For example a space aspect can react to the creation of a container and a container can react to a read-, write-, destroy-, or take-operation.

To install a new aspect in XcoSpaces and also according to the formal model of XVSM, the client application has to create an instance of that aspect and hand it over to the space along with the information to which insertion points it should be added to. Therefore only one single call is needed. However this approach has a major drawback: As the *client application* creates the instance of the aspect, it can only be added to its *local* space. Originally it was assumed that adding aspects to a

remote space was not necessary. Unfortunately the implementation of notifications proved the opposite. That caused the XVSMP to contain platform dependent information, which will be described in 4.5.3.

To overcome that issue CAPI-4 of TinySpaces demands that all aspects in the space are registered locally first before they can be installed by the local or a remote space. Consequently *two* operations have to be performed in order to add an aspect:

- First a factory has to be registered which is used to create instances of the corresponding aspect along with a name to identify it. (e.g.: “Notification1” for the factory of notification aspects) This has to be done *locally*.
- Then the aspect can be installed. This means that the aspect layer creates an instance of the requested aspect using the corresponding factory and adds it to the specified insertion point(s).

4.5.1.3 *ISpaceAspect*

This interface needs to be implemented by a space aspect. It is borrowed from the implementation of XcoSpaces and is introduced in [7]. The methods take different parameters depending on the operation and return an AspectResult object which can be used to affect the execution. For more information about how aspects can affect runtime behavior, we refer to [18].

Additionally, another insertion point has been added, which does not exist in XcoSpaces but is defined in the formal model: LookupContainer (see 37). It can be used to add additional logic to the process of looking up the ContainerReference of a named container.

One property which seems to be misplaced is the “Space” property as it stores a reference to the synchronous API of the space in which the aspect resides. That is because some types of aspects may need to modify the contents of the owning- or a remote space in order to achieve their task. A notification aspect (see 70) and aspects which add replication functionality to a space are examples for that type.

Unfortunately this approach breaks the layered architecture as it allows aspects to access higher layers as shown in Figure 34. The formal model therefore specifies, that aspects may only access the runtime layer and have to handle the creation of request messages, which is otherwise done by the API, themselves. However providing the XVSM API to aspects eases their implementation for developers and reduces the code-size as code is reused. Therefore this solution was chosen for TinySpaces. XcoSpaces follows the same approach as the layered architecture of XVSM had not been specified at the time it was implemented.

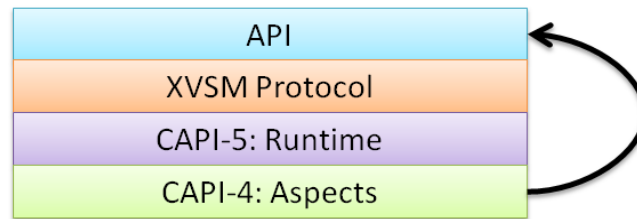


Figure 34 - Aspects accessing the API

4.5.1.4 SpacelPoint

This is an enumeration of space level insertion points. The names of its items have been borrowed from XcoSpaces. It is used as parameter for the method “InstallSpaceAspect” to specify the points where a space aspect shall be added.

In XcoSpaces the underlying type of the SpacelPoint enumeration is a 32 bit integer value. Consequently each item of the enumeration is represented by its own 32 bit integer. Therefore if an aspect (for example a logging aspect) shall be added to all 15 defined space level insertion points, which is the worst case, $15 \cdot 32 (=480)$ bits are needed to express this. As this wastes a lot of memory and network bandwidth it is especially not suitable for resource-constrained devices.

Thus, the enumeration was converted to a flag enumeration for the use in TinySpaces. Consequently each of its items is identified by a single bit which specifies if the assigned item is set (true or false). Therefore a single 32 bit integer can be used to express 32 different insertion points. However only 15 insertion points are defined at space level wherefore a 16 bit integer is absolutely sufficient.

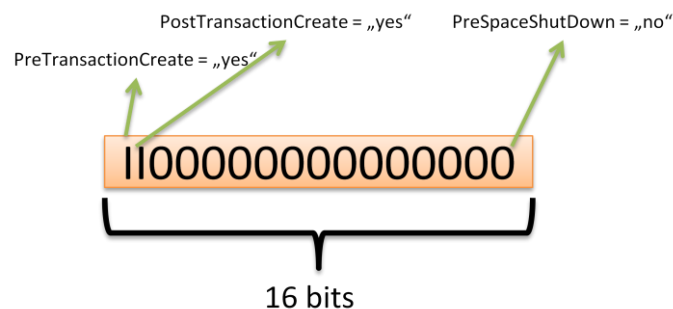


Figure 35 - Using a 16 bit flag enumeration for SpacelPoint

Considering the former example this approach saves $(480-16)$ 464 bits!

4.5.1.5 ISpaceAspectFactory

The ISpaceAspectFactory needs to be implemented by factory objects which are used to create an instance of one specific type of space aspect.


ISpaceAspectFactory	
 CreateInstance	Takes an array of arbitrary objects as parameter. Returns a new instance of the corresponding space aspect.

Table 20 - CAPI-4: ISpaceAspectFactory

„CreateInstance“ can take an array of arbitrary objects as parameter, because some aspects might need some initial information provided when they are created. For example the notification aspect needs a ContainerReference as argument. This is explained in 4.5.3.

4.5.1.6 *IContainerAspect*

As the contract for space aspects, IContainerAspect is also borrowed from XcoSpaces and is thus not further described here. For more information we refer to [7].

4.5.1.7 *ContainerIPoint*

This is an enumeration of insertion points at container level which is used as parameter for the method “InstallContainerAspect” to specify the points a container aspect should be added to.

As the SpaceIPoint enumeration the values are borrowed from XcoSpaces, but it has been converted from an enumeration of 32 bit integer values to a 16 bit flag enumeration to save memory and network bandwidth.

4.5.1.8 *IContainerAspectFactory*

The IContainerAspectFactory contract is implemented by factories which are used to create instances of container aspects.



 IContainerAspectFactory	
 CreateInstance	Takes an array of arbitrary objects as parameter. Returns a new instance of the corresponding container aspect.

Table 21 - CAPI-4: IContainerAspectFactory

4.5.1.9 *OperationContext*

The OperationContext can be used to carry additional information for all aspects, like for example the login information of the current user. As this is another contract borrowed from XcoSpaces, it will not be further described here.

4.5.1.10 *AspectResult*

This contract is also borrowed from XcoSpaces. It is returned by all methods defined in ISpaceAspect and IContainerAspect and can be used by the aspects to affect the runtime behavior:

- If OK is returned the current operation is continued without any changes.
- If SKIP is returned the following aspects need to be skipped, and in case a pre-aspect returned this value, the operation itself is also skipped. Therefore a complete operation could be replaced by a pre-aspect.
- If RESCHEDULE is returned the current operation needs to be delayed by the runtime and later re-run.

As opposed to the formal model of XVSM, there is no return value of “NOTOK” which specifies that an error has occurred within the aspect. In fact the aspect layer of TinySpaces catches any exceptions thrown by aspects and creates a “NOTOK” response itself in that case.

4.5.2 Implementation

This chapter will give an overview of the current implementation of CAPI-4 for TinySpaces.

4.5.2.1 AspectFacade

Although container aspects are defined in the formal model and provided by XcoSpaces and MozartSpaces they were omitted in the implementation of TinySpaces, which solely supports space aspects for now. The reason for this is that container aspects are more complicated to implement than space aspects, because the aspect layer needs to separate them into groups belonging to a single container. Moreover the operations of the space which will be used most are Read, Write, Take and Destroy. Therefore they need to be as fast as possible and supporting container aspects would slow them down as, each time one of these operations is called, the aspect layer has to search for container aspects that have to be called before or afterwards (pre- and post-aspects).

The aspect layer offers the runtime (CAPI-5) access to the lower layers by wrapping their functionality. Whenever a method is called, it searches for pre-aspects for the particular operation. If there are any, they are executed and afterwards the operation itself is called. Finally the corresponding post-aspects are searched and called if present.

4.5.2.1.1 Performance Considerations

Basically the aspect layer has to perform two searches for each XVSM operation (CreateContainer, Read, Take, etc.), in case the execution of each aspect and the XVSM operation itself results to “OK”: The first search has to be done to get the pre-aspects and the second to get the post-aspects. These search operations need to be optimized.

In XcoSpaces, the space aspects are grouped into lists by their insertion point. These lists are stored in an array using the integer values of the insertion points as indexes.

As TinySpaces use flag enumerations, this approach is not applicable. Furthermore it was denoted in 4.2.2.3.1 that the decision was made to use “ArrayLists” as Microsoft is going to re-implemented them in native code in the next release of the .NET Micro Framework. Consequently searching in large lists may result in poor performance, as each item of the list has to be visited to determine if it meets the search criteria. Therefore a registry for space aspects was implemented which minimizes the search effort by keeping a separate list of aspects for each insertion point in a separate property. This way the AspectFacade simply has to read the property value of the corresponding ipoint to obtain the aspects.

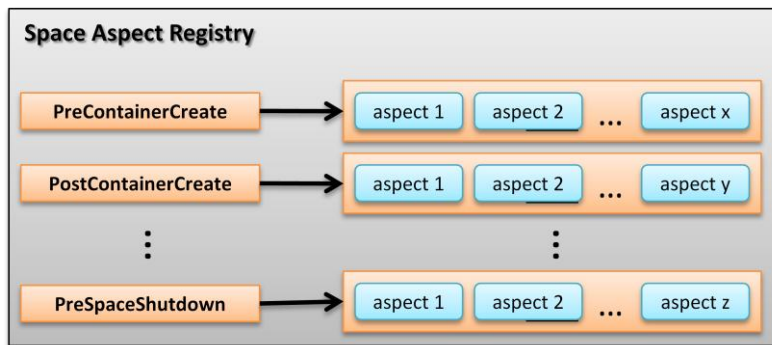


Figure 36 - Space Aspect Registry

A drawback of this approach is that code-size is raised slightly, as fifteen properties containing lists are implemented in the registry.

4.5.3 Notifications

Embedded devices are typically reactive. This is why TinySpaces support of notifications is intrinsic. Basically notifications are used to advise a client application about changes made to a container. As written in [7 S. 102] notifications can be implemented in different ways:

- A notification can inform about a change without providing the entries concerned. For example it solely carries the information that entries have been read from a container. Therefore the client application needs to get the entries itself, but on the other hand can also ignore that event. If those events are ignored frequently, this approach can save bandwidth. For this reason the approach could be applicable for TinySpaces. However, it brings a big disadvantage: In some cases it may be impossible for the client application to retrieve only those entries that caused the event, as a container might be coordinated by a FIFO-Coordinator for example. Moreover another application could already have removed those entries from the container.
- In the second approach the notification returns the concerned entries thereby providing any information needed by the observer and making additional calls unnecessary.

Concerning all the pros and contras, the decision was made to use the second approach.

Notifications in TinySpaces are implemented using aspects as in XcoSpaces and MozartSpaces. The main reason for this is that they can easily be omitted in scenarios where they are not used thereby reducing code-size.

The principle of the implementation is similar to the one which can be found XcoSpaces (see [7 S. 102]). Once the client application uses the API to create a notification, three steps are taken:

- First, a notification container, coordinated by a FIFO-Coordinator, is created in the space of the observed container.
- Second, an aspect is installed in that space and depending on its configuration, it notices when entries are read, written, taken or destroyed. This information is then written to the notification container. Additionally, the aspect needs to notice when the observed container is destroyed in order to also destroy the notification container as it is not needed anymore.

- Third, a Notification object is created. It starts a thread which periodically makes a blocking take on the notification container to retrieve the changes sighted by the aspect. This object is then returned to the client application and acts as an interface providing methods to pause and continue the notification process.

The way notifications work is simplified illustrated in the following figure.

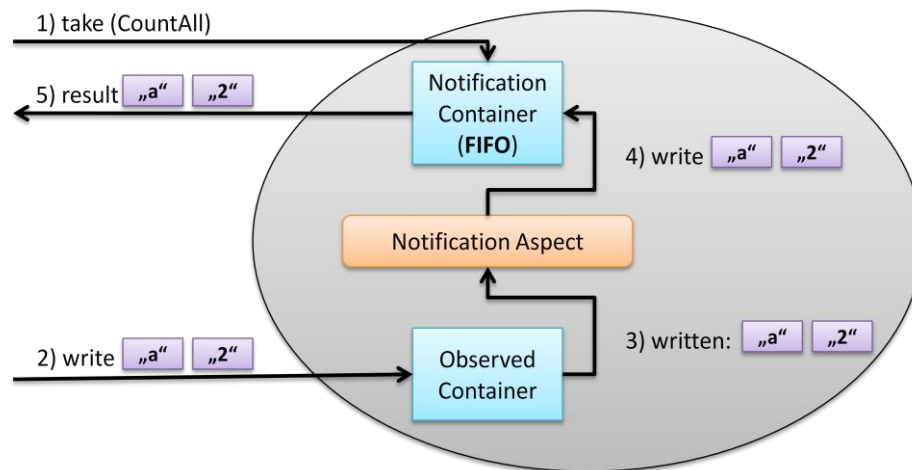


Figure 37 - Notification process

In XcoSpaces a container aspect is used to provide this functionality. This aspect implements the points of Write, Shift, Read, Take and Destroy. Thus whenever a transaction modifies the contents of the observed container, the aspect writes those changes to the notification container using the same transaction. That guarantees that those entries are only visible if the transaction eventually commits and removed if it is rolled back.

However, this approach seems to be inappropriate for resource-constrained devices, because container aspects would be needed which would decrease performance. This also imposes a lot of round trips through the runtime. (See Figure 34)

Consider for example that a client application wants to be notified about any changes (Read, Write, Take, Destroy) made on a container *c1*. When a transaction *t1* writes four entries into *c1* one by one, and also takes six entries individually, this affects that the notification aspect makes ten single write-operations on the notification container and in turn causes ten round trips through the runtime.

For the reasons mentioned above, another approach was used for implementing notifications in TinySpaces.

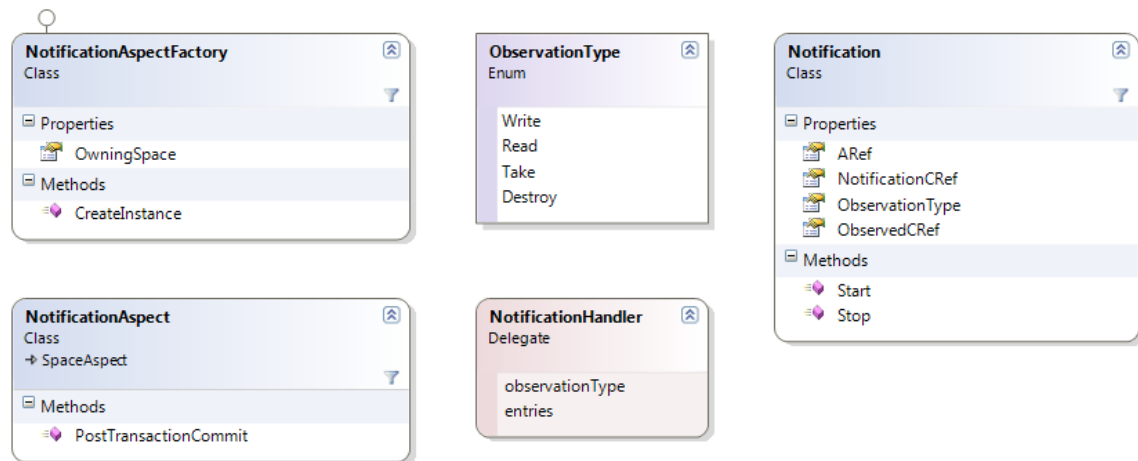


Figure 38 - Notification implementation of TinySpaces

In contrast to XcoSpaces, the notification aspect is based on a *space aspect* and makes use of the meta-information provided by a committed transaction. Thus it only implements the post-ipoint of “TransactionCommit”. To use this aspect, its factory, the NotificationAspectFactory, has to be registered with a name (e.g. “XVSM.Noification1”).

Then the operation “InstallSpaceAspect” is called by the client application, passing a reference to the targeted container, a reference to the notification container and an “ObservationType” object as parameters. The latter specifies which operation(s) shall be observed by the notification aspect. For the reasons mentioned in 4.5.1.4 this is a flag enumeration with *byte* as base type. Therefore only eight bits are used. The aspect layer then creates an instance of the aspect using the NotificationAspectFactory, handing over the three arguments.

Whenever a transaction commits, the notification aspect scans the ITransactionCommitInfo object, which is provided as a parameter for the aspects method, for modifications made on the observed container depending on its configuration (written-, read-, taken-, destroyed entries). If it finds relevant changes, it writes them to the notification container with a single write-operation for each distinct observation type (Read/Write/Take/Destroy) using a single transaction. Moreover, it checks whether the transaction that committed destroyed the observed container. In that case the aspect also destroys the notification container.

The notification feature is not included in the API of TinySpaces. Therefore, if the client application wants to create a notification, it has to create a new “Notification” object and call its “Start” method. The constructor of this class requests a NotificationHandler, which is a delegate (a kind of managed function pointer). This handler is called whenever a result is received from the notification container. Once the notification is eventually not needed anymore, the client application solely has to call the “Stop” method, in order to remove the notification aspect and the notification container.

This new approach reduces the amount of write-operations and the resulting round trips through the runtime layer as only one operation is needed per transaction and per “ObservationType”. Thereby it

reduces the amount of created transaction logs, which unloads the heap as well as the garbage collector thus raising runtime-performance.

However, the approach has a major drawback. Whenever a transaction commits *each* notification aspect is called and in turn searches for suitable changes made on container, even if the container it is watching has never been touched by that particular transaction. For that reason, the approach is not applicable for the enterprise versions of XVSM, namely XcoSpaces and MozartSpaces.

4.5.3.1 Technology Dependence of Aspects in MozartSpaces and XcoSpaces

As already denoted, the formal model of XVSM specifies, that aspects are added to an insertion point in a single step. The problem that emerges from that fact is that the type of the notification aspect has to be sent over the network, if it should be added to a remote space. This in turn has to create an instance of that aspect and add it to the specified insertion point(s). Once an XML-based technology independent protocol was developed to provide interoperability between XcoSpaces and MozartSpaces, this was a serious problem as type information is always tightly coupled to a specific technology and should therefore not be present in that protocol.

Consider for example that a client running XcoSpaces (.NET) wants to get notified about changes of a container which resides in an instance of MozartSpaces (Java). To accomplish this, the technology independent protocol is used and the client space needs to know which technology the space, which it wants to add an aspect to, is based on and send the appropriate type information. The XML-based message looked similar to the following figure. (Figure 39)

```
<addAspect type="java" containerReference="TcpXML://localhost:9876/containers/2">
  <value>com\mindprod\mypackage\NotificationAspect.java</value>
  <ipoints>...</ipoints>
  <properties>...</properties>
</addAspect>
```

Figure 39 - Adding an aspect implemented in JAVA using XVSM

As can be seen, the full class path of the notification aspects implementation class needs to be contained and the “type” attributes value is “java” to specify that the aspect is implemented using that technology.

If, on the other hand, an aspect shall be added to XcoSpaces using XVSM, the message would look differently as shown in the following figure. (Figure 40)

```
<addAspect type="dotNet" containerReference="TcpXML://localhost:9876/containers/2">
  <value>XcoSpaces.Aspects.Implementation.NotificationAspect, XcoSpaces.Aspects</value>
  <ipoints>...</ipoints>
  <properties>...</properties>
</addAspect>
```

Figure 40 - Adding an aspect implemented in Microsoft .NET using XVSM

This time the “type” attribute contains “dotNet” specifying that the aspect is implemented using the .NET Framework. The value now contains the full name of the implementation type along with the assembly name in a format specific to .NET.

This is obviously no clean approach as it requires all existing space implementations to be adjusted along with the protocol as soon as a new implementation of a space, based on another technology, needs to be supported. For example: TinySpaces.

Therefore, the registration step has been introduced (see 4.5.1.2). With this solution, each space simply registers its implementation of the notification aspect under a well known name. This name is then used as the only information sent over the network along with the insertion points, thus making it unnecessary to send type information in order to install an aspect as the following figure shows. (Figure 41)

```
<addAspect containerReference="TcpXML://localhost:9876/containers/2">
  <value>NotifiationAspect</value>
  <ipoints>...</ipoints>
  <properties>...</properties>
</addAspect>
```

Figure 41 - Adding an aspect in general using the technology independent approach

An in depth look at the current definition of XVSMP in MozartSpaces can be found in [8 S. 61-70].

4.6 CAPI-5: Runtime

This chapter provides an in dept look at the contracts of the runtime layer as well as its current implementation in TinySpaces.



4.6.1 Contracts



Figure 42 - CAPI-5: Contracts

4.6.1.1 *IRuntimeFacade*

This interface is used by the API to communicate with the lower layers.

IRuntimeFacade		
	Process	Takes a request (e.g.: a "WriteRequest") as parameter.
	ResponseReceived	This event is fired whenever a response is received from the local space or a remote one. The handler is of type ResponseReceivedHandler.

4.6.1.2 *ResponseReceivedHandler*

This delegate takes a response as parameter and is used by the API to subscribe to the "ResponseReceived" event of the runtime façade.

4.6.2 Implementation

As can be seen, there are no contracts for the building blocks of the runtime like the one handling timeouts or the one handling the rescheduling of requests. The reason for this is that the runtime can have various forms. For example, a runtime for extremely resource constrained devices can be developed which, for that reason, is single threaded and thus has a much simpler logic for rescheduling requests. In that case the original contracts do not match anymore, but as the inner building blocks are never visible to the enclosing layers using the current layered approach, those contracts are not necessary for TinySpaces.

The runtime of XcoSpaces (see 1.6.6) is optimized for concurrency. As thread synchronization mechanisms are costly, the number of these synchronization points is minimized in TinySpaces and the trade-off (reduced concurrency) is accepted. Consequently only the CoreProcessor is

multithreaded and its worker threads perform all tasks required for an operation sequentially (Read/Take/Write/CreateContainer/...), including updating the TimeoutHandler and the WaitHandler. Moreover requests targeting a remote space are sent via the Communication Core using the user applications thread.

Although using multiple threads within the CoreProcessor requires additional computation power as multithreading is only simulated by the Tiny CLR, it is required for aspects though. Consider the notification aspect introduced in 4.5.3 as an example. Whenever a transaction is committed, which modified the container that is observed, the aspect writes the corresponding entries to the notification container. To achieve this it uses the synchronous API, which blocks the calling thread until a result is returned. As the aspect itself is executed by a thread of the CoreProcessor, it would be blocked and therefore the call of the aspect would need to be processed by another thread. If only one single worker thread was used by the CoreProcessor, this would obviously cause a deadlock as shown in the following figure. (Figure 43)

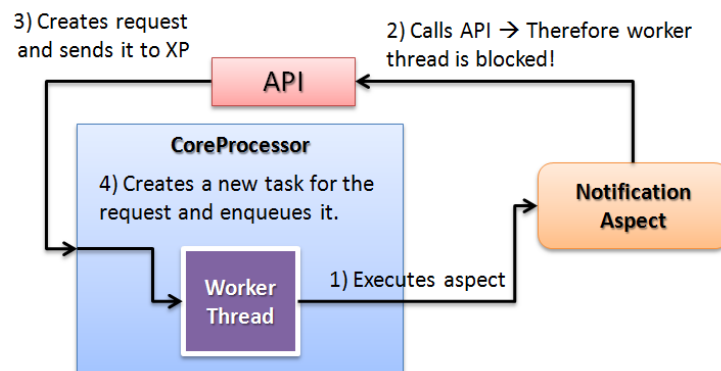


Figure 43 - Deadlock caused by aspect with single-threaded CoreProcessor

The components of TinySpaces' runtime and how the API is involved is depicted in Figure 44.

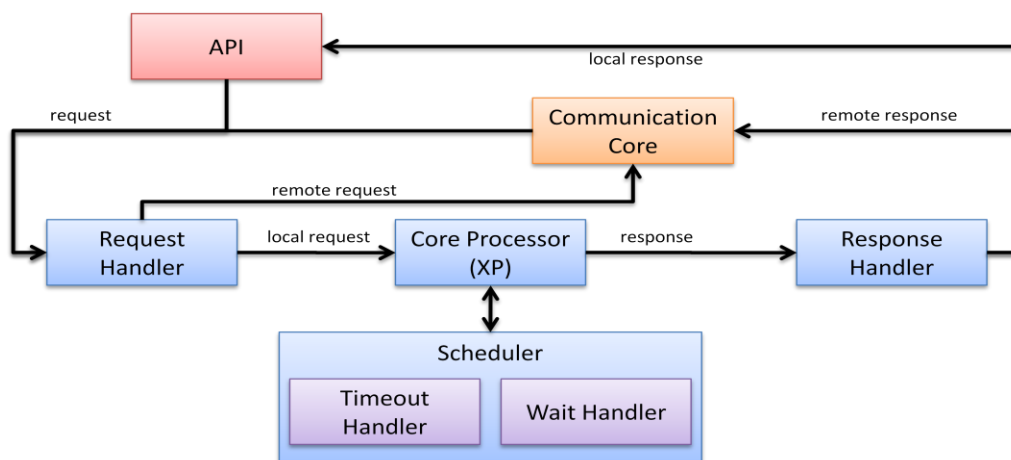


Figure 44 - TinySpace Runtime

Whenever a request is received from the API or the CommunicationCore, it first is handled by the RequestHandler, which can determine whether the request targets the local or a remote space. In case the local space is targeted, it forwards the request to the CoreProcessor. Otherwise the request is sent to the CommunicationCore which transmits it to the targeted space.

If the CoreProcessor gets the request, it processes it and informs the Scheduler about the results of the requests operation. The Scheduler's task is to synchronize the Timeout- and the WaitHandler and to forward their results to the CoreProcessor.

The TimeoutHandler determines the expiration time of requests and fires an event at such an occasion, which is further processed by the CoreProcessor.

The WaitHandler, on the other hand, stores delayed requests and reschedules them, whenever an appropriate event is received from the CoreProcessor.

Once the CoreProcessor has processed a request, it generates a response containing the desired result, or an error, which occurred during processing. It then forwards this response to the ResponseHandler which in turn is able to determine, whether it has to be sent using the CommunicationCore, or can be forwarded to the local API.

4.6.2.1 CoreProcessor

The CoreProcessor contains a nested class called XPTask. This contains the complete logic for processing an incoming request. Its purpose is to store the required information and process the request in a separate thread. Therefore the CoreProcessor creates a new XPTask for each request, and enqueues this task in the CommonThreadPool, which uses lazy thread creation: When a task is enqueued, the thread pool determines if there is any idle thread and wakes it up. Otherwise it starts a new thread, but only if the maximum thread count specified is not reached.

4.6.2.1.1 Implicit Transactions

When a thread runs the "Execute" method of the XPTask, it determines if the request to process requires a transaction in order to succeed. If that is the case and no TransactionReference is contained in the request, the AspectFacade is used to implicitly create a new one, which is then attached to the request.

Then the request itself is worked off. An important point here is that the runtime does not know how the requests have to be processed. They contain the logic to execute themselves. Therefore the XPTask solely calls their "Execute" method providing the façade of CAPI-4 as parameter (IAspectFacade). This is further explained in 4.6.2.2.

Thereafter, it is determined whether an implicit transaction has been used for this request, and in this case that transaction is committed

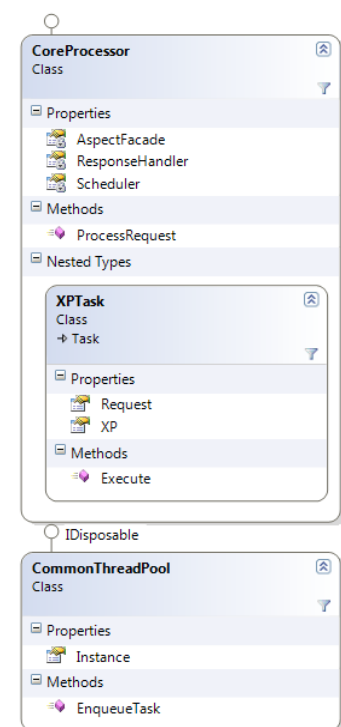


Figure 45 - CoreProcessor

immediately in case the operation succeeded. After each of these three operations, their results are forwarded to the Scheduler to update the Wait- and the TimeoutHandler.

Finally a response is created which is again performed by the request objects themselves. The XPTask solely calls their “CreateResponseContent” method and hands over the results of their “Execute” method.

The only request the runtime handles itself is the shutdown request as it is the only one that targets the runtime and not any lower layer (CAPI-4 to CAPI-1). When a shutdown request is received, the XPTask executes it as the other requests providing the façade to CAPI-4 to give the aspect layer the chance to execute pre-aspects attached to this insertion point. Thereafter, the XPTask starts the shutdown process of the runtime.

4.6.2.2 RequestMessageContent

This contract needs to be implemented by the content of each request that has to be handled by the runtime. The properties hold several timestamps which are used by the runtime:

- The “ExpireTime” is used by the TimeoutHandler. It is calculated by adding the value of the “TimeOut” property to the value of the “FirstExecutionTime” property and specifies the point in time the request invalidates.
- The “FirstExecutionTime” is only set once by the CoreProcessor and specifies when the request was processed for the first time.
- The “TimeOut” property contains a value indicating the period of time this request is valid.
- The “LastExecutionTime” property is set every time this request is processed by the CoreProcessor. It is used by the WaitHandler which will be explained in 4.6.2.4.

Furthermore, the contract defines two methods:

- The abstract “Execute” method takes an instance of IAspectFacade and is called by the CoreProcessor. It encapsulates the logic bound to one particular type of request. For example the implementation of that method in a “CreateTransactionRequest” will call the method “CreateTransaction” of the IAspectFacade object and provide all parameters the request contains. Then it will return the complete result provided by CAPI-4.
- The “CreateResponseContent” is used by the CoreProcessor to perform the second step: When the response of the “Execute” method has the status “OK” or “NOTOK”, this method is called and causes a response message to be created that can be handed over to the CommunicationCore.

As this logic is not implemented within the runtime layer itself, no conditional statements (as in XcoSpaces) are needed and therefore the code-size is reduced. Additionally, further requests can be developed without the need of modifying the runtime itself.

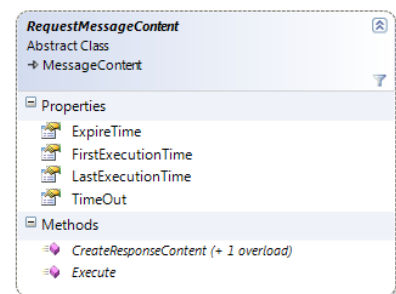


Figure 46 - RequestMessageContent

4.6.2.3 Scheduler

The task of the Scheduler is to synchronize the WaitHandler and the TimeoutHandler. Whenever an XVSM operation has been executed by the CoreProcessor the results along with the corresponding request are forwarded to those two handlers.

In case a request times out, the Scheduler removes it from the WaitHandler and forwards this event to the CoreProcessor.

If in contrast the WaitHandler fires an abort-event - this might be the case if a container, another request wants to read from, is destroyed - that request is removed from the TimeoutHandler first and thereafter forwarded to the CoreProcessor.

4.6.2.4 WaitHandler

The WaitHandler solves a complex task: It stores requests that need to be delayed and reschedules them when an appropriate event is received from the CoreProcessor.

A request is delayed in two cases depending on the result received from CAPI-4 and the lower layers.

- The status flag “LOCKED” is returned meaning that one or more containers or entries could not be accessed as another transaction has locked them.
- The status flag “DELAYABLE” is returned. This means that either a write-operation was made on a bounded container which was already full, or a read-, take- or destroy-operation on an empty container.

Therefore, the WaitHandler does not keep track of any kind of request but only of operations concerning containers (CreateContainer, LookupContainer, DestroyContainer) and operations concerning entries (Read, Write, Take, Destroy).

Then it waits for a commit or a rollback of a transaction to reschedule any of the waiting requests. The reason for this is that when a transaction commits the following happens:

- Any locks it acquired are removed.

For that reason requests that returned with the status “LOCKED” can be rescheduled.

- The changes made by the transaction to the contents of a container, like added and removed entries, become visible to all other transactions.

Accordingly, requests which did not succeed and returned with the status “DELAYABLE” can be woken up. However a distinction has to be made:

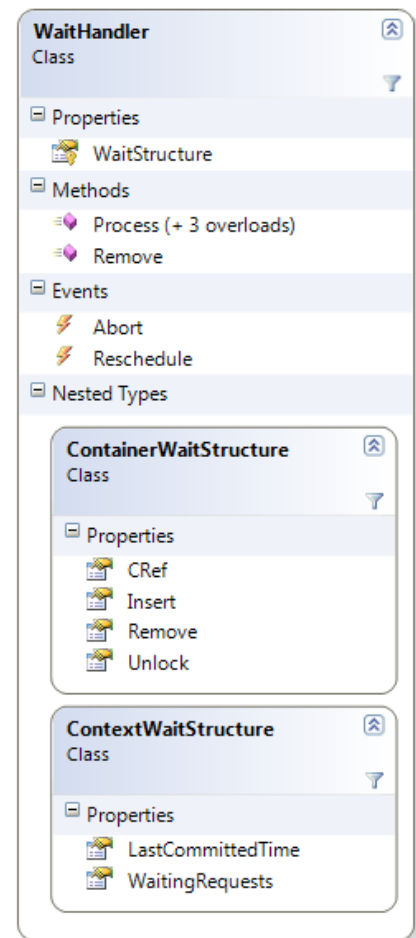


Figure 47 - WaitHandler

If a transaction added entries, only read-, take- and destroy-requests need to be rescheduled as they are waiting for entries to be added. On the other hand, a write-request only needs to be rescheduled if an entry was removed by the transaction.

The implementation of the WaitHandler was slimmed as no distinction is made between “short-term locks” and “long-term locks”. It therefore does not comply with the formal model of XVSM. This distinction is, however, not needed in TinySpaces, as the general meta-information of containers as well as the accountant information of coordinators is not modified using transactions and as a result cannot be locked. Consequently code-size and computational costs are reduced.

The WaitHandler separates incoming requests by the container they need to access using a ContainerWaitStructure object. The original idea was to create that object once a container is created, but this wastes memory and CPU time, as some containers might never be accessed concurrently by multiple transactions. For that reason, this object is created once the first request targeting that particular container is delayed.

This ContainerWaitStructure contains the ContainerReference of the corresponding container as well as three properties containing a ContextWaitStructure object for each wait category.

- The category “Insert” contains consuming requests which are waiting for entries to be written into the container. (Read-, take- and destroy-requests)
- The requests filed under “Remove” are, on the contrary, producing requests, waiting for entries to be removed. (Write-requests)
- The final category named “Unlock” contains entries, which wait for locks to be released from the container itself or any of its entries. Such requests are for example a read-request as well as the request for destroying the container.

The ContextWaitStructure contains the requests along with a timestamp: the “LastCommittedTime”. This timestamp specifies when the last transaction, that modified the particular container, was committed or rolled back. This is important as the runtime is multithreaded. Consider the following example which is depicted in Figure 48: Two requests $r1$ and $r2$ with their corresponding transactions $t1$ and $t2$ are running on two separate threads. $t1$ is executed first and locks entry $e1$ in container $c1$. Thereafter $t2$ also tries to lock $e1$, but fails as it is already locked by $t1$. Next the execution of $t1$ is continued and the transaction commits successfully. Finally, the execution of $t1$ is also continued and its request is delayed and consequently handed over to the WaitHandler. The problem that arises now is that the WaitHandler receives the event caused by the commit of $t1$ before the delayed request $r2$ is received. Therefore $r2$ will never be woken up by that event and could eventually expire erroneously.

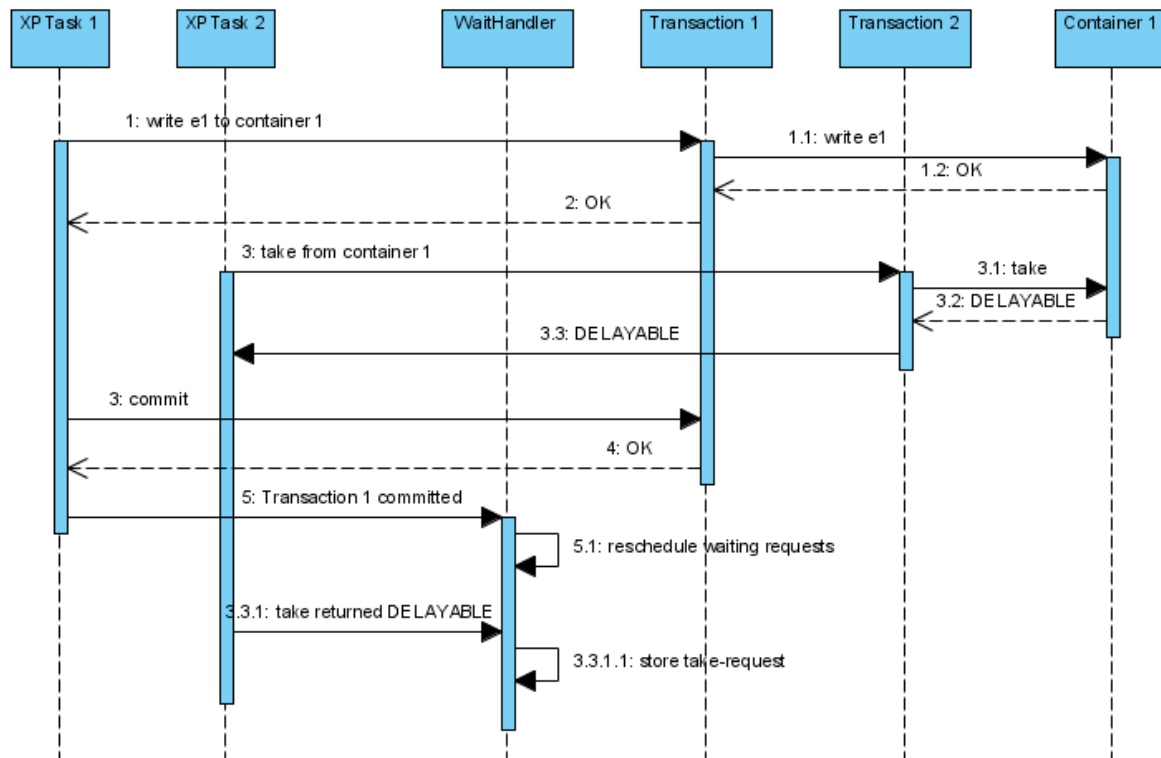


Figure 48 - Request expires erroneously

For this reason, the “LastCommittedTime” is stored and when r_2 is received by the WaitHandler, it is compared to the timestamp of r_2 , which specifies the last time it was processed by the CoreProcessor. If r_2 was processed *before* the last commit happened, it is possible that the corresponding transaction contained information, which would have caused r_2 to be rescheduled. Therefore r_2 is rescheduled immediately.

As already denoted the WaitHandler reschedules waiting requests whenever a transaction commits or rolls back and consequently has to determine which of the waiting requests shall be rescheduled. To achieve this it uses the meta-information that is provided by CAPI-2 upon a commit: The ITransactionCommitInfo. First it iterates through all the contained IContainerModificationInfo objects which contain the ContainerReference of the modified container. Consequently the WaitHandler can use that reference to it to find the corresponding ContainerWaitStructure.

If the IContainerModificationInfo contains any written entries and the ContainerWaitStructure contains requests in the “Insert” category, those requests are rescheduled. The same happens for the requests stored in the “Remove” category, if entries were destroyed or taken within the transaction. Finally, all requests filed under the “Unlock” category are rescheduled as locks have obviously been removed from entries and the container. As a final step, the WaitHandler determines whether any containers have been destroyed within the committed transaction and removes the corresponding ContainerWaitStructures in that case.

4.6.3 TimeoutHandler

The TimeoutHandler is responsible for invalidating requests and transactions that expire. Therefore it tracks their expiration times and notifies the CoreProcessor in that case. Whenever a transaction expires it has to be rolled back and all of its remaining requests need to be canceled. If, on the other hand, a single request of a transaction expires, the same steps need to be taken.

To keep track of the expiration times of requests and transactions the TimeoutHandler manages two lists:

The first one contains all requests, which are sorted by their expiration times in an ascending order. This is because a timer is used internally, which wakes up when the next request expires. It then removes the request from that list and uses the *new* first item to determine the duration to sleep.

The second one stores items which contain a TransactionReference and the requests belonging to the corresponding transaction. Once a request or the transaction itself expires, the TimeoutHandler uses that list to determine the remaining requests of a transaction, in order to remove them from the list of requests and also to inform the CoreProcessor that those requests have also become invalid. Furthermore there could be requests, belonging to a particular transaction, which have a timeout value of “0” and are therefore not stored within the list of requests, as they actually have no expiration time. For more information about “timeout 0” we refer to [18].

4.6.3.1 TimeoutHandler and Deadlocks

One reason why the TimeoutHandler is so important is that XVSM currently does not define a deadlock detection mechanism. Therefore using timeouts is the only way to overcome deadlocks in TinySpaces. Consider for example two transactions $t1$ and $t2$ trying to read two entries from a container $c1$ concurrently using different coordinators:

- $t1$ wants to take two entries from container $c1$ using two successive take operations with a FIFO-Coordinator (First-In First-Out).
- At the same time $t2$ also tries to take two entries from that container, but uses a LIFO-Coordinator (Last-In First-Out).

Additionally, the container $c1$ only contains two entries. What could happen now is that $t1$ succeeds taking the first entry, but fails to lock the second one, as it is already locked by $t2$ which, in contrast to that, cannot lock the other entry, as it has already been locked by $t1$. This can occur as the entries are still in the container, because none of the transactions has committed at that time. As these two transactions are blocking each other, a deadlock occurs. This process is depicted in Figure 49.

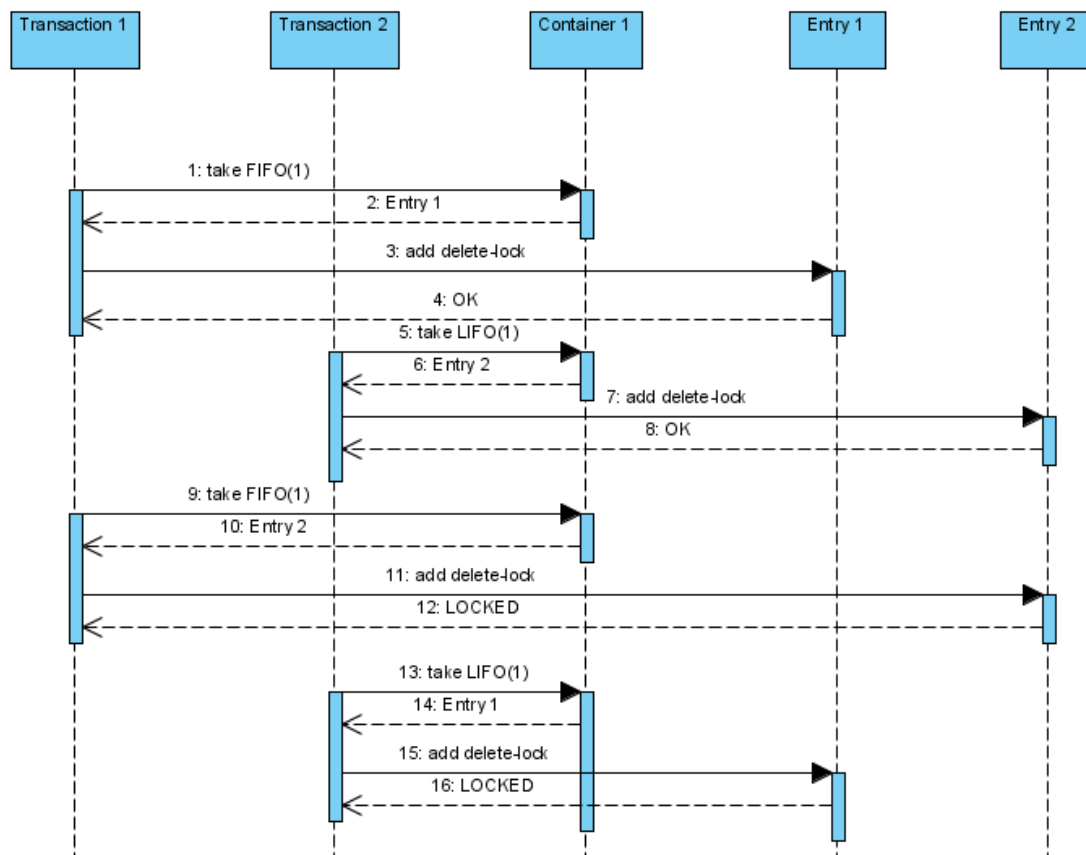


Figure 49 - Deadlock with transactions

Therefore, they would get stuck forever if there was no timeout handling. Let us assume that the second request of t_2 eventually expires. In that case the timeout handler notifies the CoreProcessor which in turn rolls back t_2 and sends an error message to the client application. This rollback is noticed by the WaitHandler which consequently wakes up the waiting request of t_1 , which will therefore succeed its task and return the second entry to the user application.

The only way that a deadlock can occur now, is when both transactions specify a timeout value of “infinite” and the requests use “timeout 0”. For this reason, applying an infinite timeout to a request or transaction is dangerous and should only be done when it is *absolutely* necessary. One case where an infinite timeout is required for example is the implementation of notifications. If the blocking takes on the notification container had a lower timeout specified, the observing application had to restart this operation periodically. This would lead to unnecessary computational costs, network traffic and power consumption and is therefore not feasible, especially for mobile embedded devices.

4.7 CAPI-5b: Communication

The communication module is completely separated from the runtime. The main reason for this approach is configurability. For example, TinySpaces could be used to coordinate mobile agents and therefore a communication module might be needed, which is capable of connecting to remote spaces automatically as soon as they come within range. Moreover, a different specialized runtime could be

developed (e.g.: a single-threaded one) for one particular device. In that case the communication module could be reused.

In contrast to XcoSpaces, TinySpaces support several transport protocols along with several serialization mechanisms at the same time. The following example, which is depicted in Figure 50, demonstrates why this feature is absolutely necessary for TinySpaces: Three embedded devices (*A*, *B* and *C*) are communicating with each other via ZigBee. [54] This is a wireless technology which allows for data rates of up to 250 Kbit/s. Therefore the data sent needs to be as compact as possible wherefore the .NET Micro Framework binary serialization is used. Additionally device *C* is communicating with device *D*, which is a personal computer and therefore runs a full operating system (and not the .NET Micro Framework). Consequently it does not support that binary serialization format. For this reason an interoperable format, like an XML-based one, needs to be used. Furthermore personal computers generally do not provide a ZigBee module and therefore device *C* and *D* communicate over Ethernet (TCP/IP).

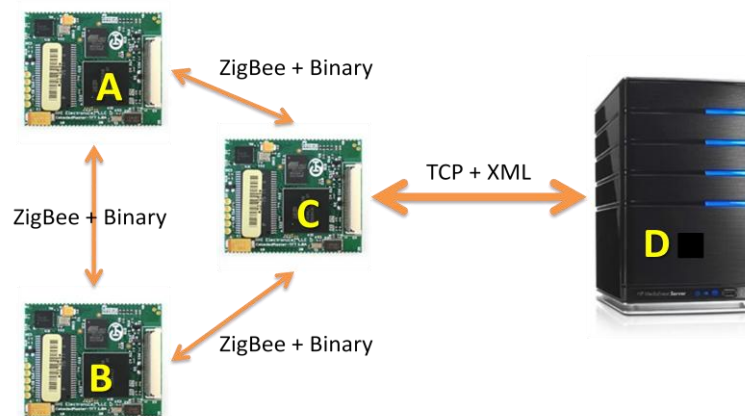


Figure 50 - Example scenario with multiple transport protocols

Due to the layered architecture of XVSM, CAPI-1, -2 and -4 cannot set the address of entity references (ContainerReference, TransactionReference and AspectReference) they create. For example, if a new container is created, CAPI-1 solely assigns the locally unique ID to the ContainerReference that it returns. This is because CAPI-1 cannot know where the created ContainerReference is sent to: It could be used by a local client program, but also sent to a remote space using one of the communication services the communication module provides and the address of an entity reference depends on the communication service used to send it.

It is therefore the communication modules responsibility to set the address of all entity references which are sent to another space:

- When a response is sent using a service, the local address of that service is used as the address of each entity reference contained in that response.
- If a request is received on the other hand, the address is removed from the entity reference if it matches the local address of the receiving service. This is important as the lower layers (CAPI-1, -2 and -4) also take the address property into account to identify an entity. If a local entity is to be referenced, the address property needs to be empty.

This chapter focuses on the contracts and the implementation of the communication module.

4.7.1 Contracts

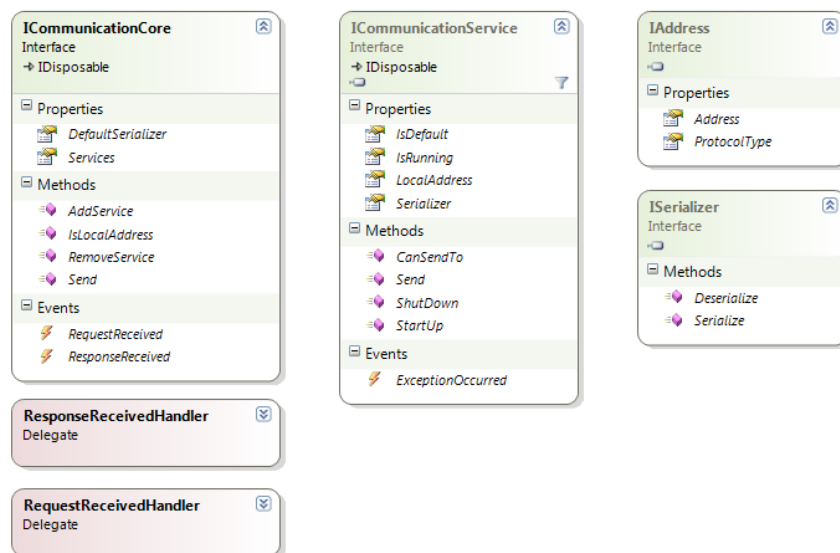


Figure 51 - CAPI-5b: Contracts

4.7.1.1 ICommunicationCore

This interface is used by the runtime to communicate with other spaces. The CommunicationCore is responsible for handling multiple communication services along with their serializers and for adapting the address property of outgoing and incoming entity references.

ICommunicationCore		
	DefaultSerializer	Gets/Sets the serializer which shall be used if the communication services do not provide one.
	Services	Returns all communication services which are currently managed by the communication module.
	AddService	Adds a communication service.
	RemoveService	Removes a communication service.
	IsLocalAddress	Takes an IAddress object as parameter and returns true if it is the local address of one of the communication services.
	Send	Takes a Message object as parameter and sends it to its destination.
	RequestReceived	This event is fired whenever a Request object is received by a communication service. The method signature is defined by the "RequestReceivedHandler" contract.
	ResponseReceived	This event is fired whenever a Response object is received by a communication service. The method signature is defined by the "ResponseReceivedHandler" contract.

Table 22- CAPI-5b: ICommunicationCore

4.7.1.2 *ICommunicationService*

This contract needs to be complied with by an implementation of a communication service.











 ICommunicationService		
	IsDefault	Specifies whether this is the default service of the communication module.
	IsRunning	Specifies whether the service is currently active.
	LocalAddress	Returns an IAddress object specifying the local address of that service.
	Serializer	Returns the ISerializer object that shall be used to (de)serialize messages for that service.
	CanSendTo	Takes an IAddress object as parameter and returns true if the service is able to send data to that address.
	Send	Takes a byte array along with an IAddress object as parameter and sends it.
	StartUp	Starts the service.
	ShutDown	Stops the service.
	ExceptionOccurred	Is fired whenever an exception occurs within the service or one of its worker threads.

Table 23 - CAPI-5b: ICommunicationService

4.7.1.3 *IAddress*

The IAddress contract is used to distinguish between the different addressing schemes used by transport protocols. As already denoted (4.2.1.1) in TinySpaces addresses are, in contrast to XcoSpaces and the formal model, not expressed with strings but with address objects.




 IAddress		
	Address	This property contains an arbitrary address object depending on the transport protocol. For example if TCP/IP is used this could be an URL, and if ZigBee was used, it could be a 16 bit integer.
	ProtocolType	This property contains an identifier for the protocol/communication service that shall be used to send the message.

Table 24 - CAPI-5b: IAddress

4.7.1.4 *ISerializer*

This contract is used by all serializers, like XML or Binary just to give some examples.




 ISerializer		
	Serialize	Takes an arbitrary object along with the object's type as parameter and returns a byte array.
	Deserialize	Takes a byte array along with the object's type as parameter and returns the de-serialized instance.

Table 25 - CAPI-5b: ISerializer

4.7.2 Implementation

4.7.2.1 Built-in Communication Services

Currently only one communication service for TinySpaces is implemented, which makes use of sockets to send data using the TCP/IP protocol. The way it works is trivial and is omitted for this reason.

4.7.2.2 Built-in Serialization Mechanisms

Three different serialization modules were implemented for TinySpaces.

4.7.2.2.1 .NET Micro Framework Binary Serialization (abbreviated *Binary*)

This serialization module uses the built in support of the .NET Micro Framework for binary serialization. As this feature is implemented in native code, this module is by far the fastest one and has the least code-size. Additionally it produces the smallest data packets. However, the drawback is that this serialization mechanism is not compatible with any other platform. Consequently it can only be used to send data to devices which run the .NET Micro Framework. More information along with a comparison is provided in 5.4.

4.7.2.2.2 .NET Binary Interoperable Serialization (abbreviated *Binary Interop*)

This serialization mechanism creates a binary format which can be used to send data to spaces that run the full .NET Framework and/or the .NET Compact Framework. To achieve this, type information has to be included in the messages, and therefore they are larger than those of the first mentioned serialization mechanism. As this is a custom implementation in managed code, its code-size is also larger. Additionally it is heavily relying on Reflection which makes this mechanism slower than *Binary*. More information along with a comparison is provided in 5.4.

4.7.2.2.3 XML Serialization (abbreviated *XML*)

The format this XML serialization mechanism uses is not compatible to the current XVSMP. The main reason for this is that this protocol is outdated, as it does not apply to the layered architecture of XVSM anymore.

One reason for this is that each response is returned using a write operation whose entry contains the response message. Therefore, the receiving communication core needs to search through the entries of each received write request to determine whether it is indeed a simple write request or a response from a remote space. This indirection raises the message size unnecessarily as in most of the current applications a **virtual** answer container is specified. Furthermore, this protocol neither supports the creation of named containers nor looking up a well known named container. Also the two step approach concerning aspects (refer to 4.5.1.2) is currently not supported. Therefore, a different XML format was chosen which will not be described here as that would go beyond the scope of this thesis.

The big advantage of the XML serialization is that it is completely platform independent. However, it creates comparatively large messages. Additionally, as Reflection of the .NET Micro Framework does

not support the manipulation of getting and setting properties of objects at runtime, it has to be implemented completely manual and for each class separately. For this reason, this serialization mechanism has by far the largest code-size. More information along with a comparison is provided in 5.4.

5 Benchmark

To determine how TinySpaces perform on the targeted devices, measurements concerning performance, memory usage and energy consumption have been conducted. The results are offered and explained in this chapter. Unfortunately no comparable middlewares exist for the .NET Micro Framework which could be used as a reference, wherefore the results can only be used to argue whether TinySpaces scale well, to determine possible bottlenecks concerning the performance, and to get an idea which dimensions of entries and containers TinySpaces can handle.

For these tests version 3 of the .NET Micro Framework has been used; Version 4.0 has been released in November 2009, but is not supported by any devices yet.

The device used for these tests is the ChipworkX Development System V1.2 by GHI Electronics, LLC [55] as it offers TCP/IP communication out of the box and is one of the most modern devices available. A development system is a board which already provides most of the interfaces the embedded module supports. That module is the ChipworkX Module which provides a 200 Mhz 32-bit ARM 9 Processor along with 64MB SDRAM.

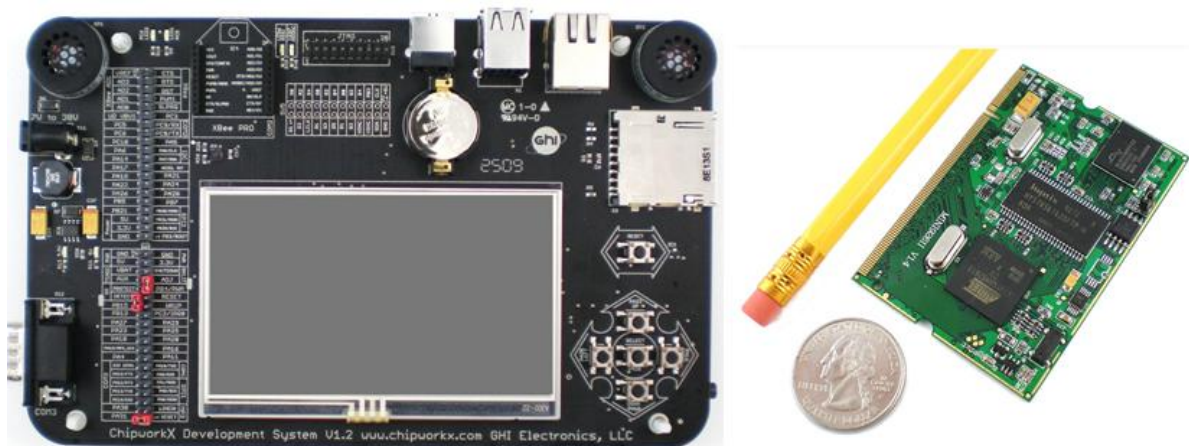


Figure 52 - ChipworkX Development Sastem V1.2² (left) and ChipworkX Module³ (right)

5.1 Performance Benchmark

The most frequently used operations of a space are write-, read-, take- and destroy-operations. As the take-operation is very similar to the destroy- and read-operation, but is the most expensive one, only write- and take-operations are benchmarked. For each coordinator (FIFO and Key) twelve tests are

² <http://www.ghielectronics.com/images/products/GHI-00125-large.jpg>

³ <http://www.ghielectronics.com/images/extras/ChipworkX-SIZ-large.JPG>

performed, in which 10, 100 and 500 entries are written to a single container, either using a separate implicit transaction for each operation or one single explicit (user) transaction, and are executed on a single thread at API level. The results show the average measurements resulting from ten test-runs and are depicted in the following figure. (Figure 53) The y-axis contains the average duration of each test measured in seconds. The x-axis contains the six test groups, whereby the name of each one is a combination of the coordinators name and the number of entries written. (e.g.: “FIFO 10” is the name of the test run where ten entries are written using a FIFO-Coordinator) The blue bars depict the results achieved using implicit transactions and the red bars those achieved using one explicit user transaction.

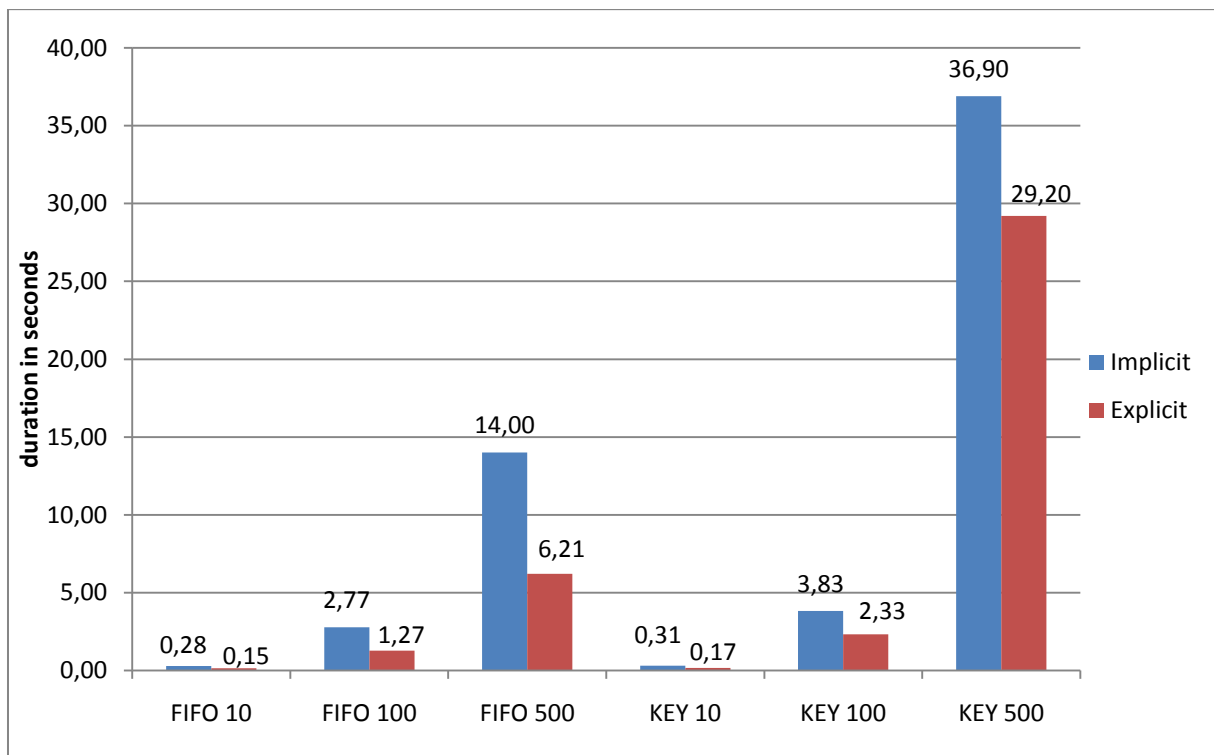


Figure 53 - Write benchmark

As can be seen the FIFO-Coordinator performs better than the Key-Coordinator in general. This is because it does not rely on a hash table like the Key-Coordinator, which has been implemented in managed code (C#) and causes higher computational costs for inserting and removing items.

Furthermore, the blue bars are generally higher than the red ones. This shows that operations running within one single transaction (explicitly) perform better than those relying on implicitly created transactions. That is because implicit transactions are created and committed by the runtime for each single write-operation separately. Therefore, two additional operations have to be performed and that effect higher computational costs.

The results show that the FIFO-Coordinator scales linearly, while the Key-Coordinator does not. This can be traced back to the performance of the hash table used by the Key-Coordinator internally.

The tests to benchmark take-operations were performed in a similar fashion: Again a container coordinated by either a FIFO- or Key-Coordinator was utilized, using either one explicit transaction for all operations, or implicit transactions for each separate operation. Moreover, three tests differing in the number of entries that is taken separately from that container were performed per coordinator. The following figure (Figure 54) shows the results. Again the y-axis shows the average duration of each test in seconds, and the x-axis contains the six test groups named using the name of the utilized coordinator and the number of entries taken. The blue bars stand for the tests performed using implicit transactions and the red ones for those performed using one single explicit transaction for the whole test run.

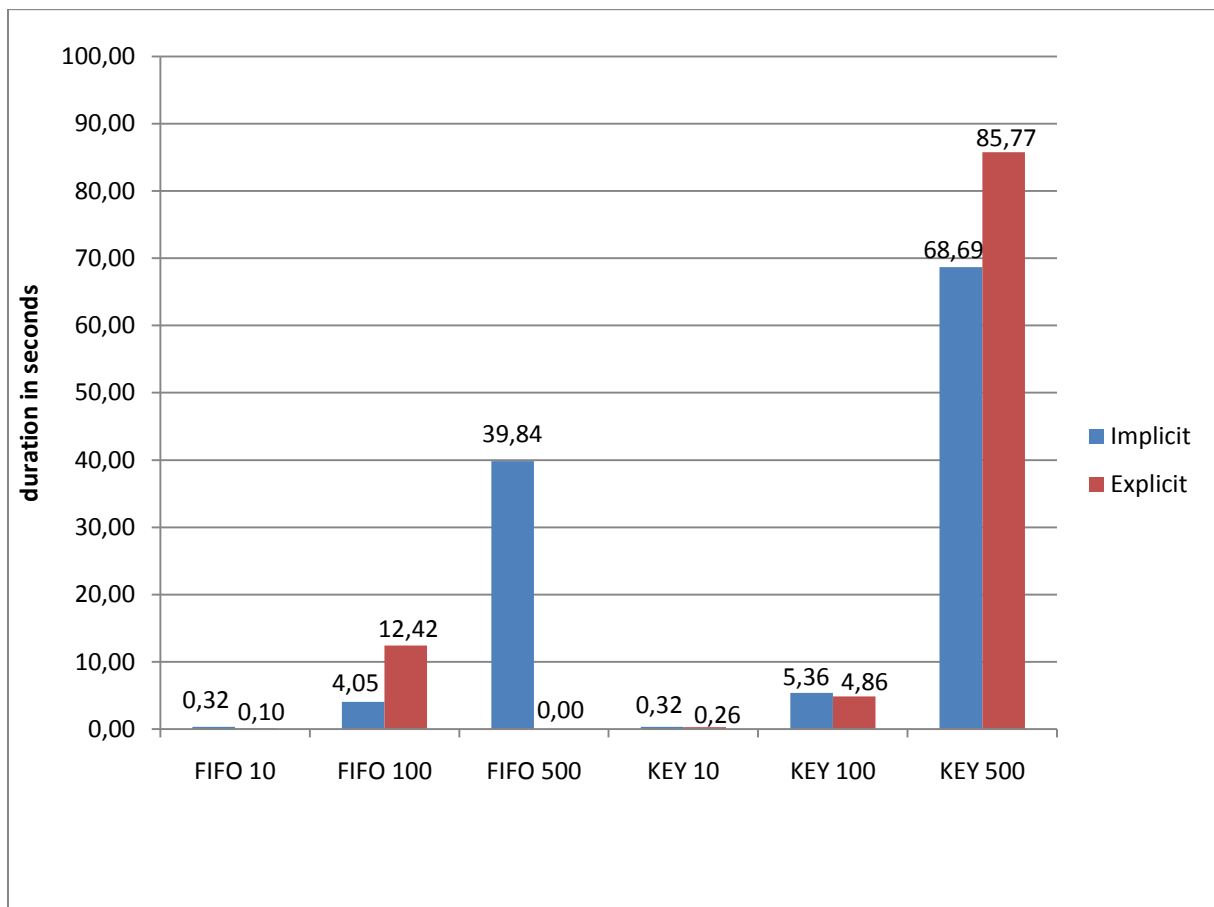


Figure 54 - Take benchmark

When a small amount of entries (about ten) is contained, take operations perform similar to write-operations. However, as opposed to the write-operations, take-operations scale much worse. One important reason for this is that at least two queries have to be performed as opposed to the write operation: The query of the transaction layer which filters all invisible items and the query of the corresponding coordinator. To overcome that issue it might be possible to enhance the coordination queries, so the transaction layer can ask them to propose entries instead of running through all entries in a container, filtering out the invisible ones and passing the result set to the coordination query. However, that approach is still under development.

The results show that, as opposed to write-operations, using one single explicit user transaction for all take-operations results in worse performance than when the runtime creates and commits an implicit transaction for each operation separately. The reason is the following: Implicit transactions are committed immediately after a take-operation succeeds. Consequently the taken entry is physically removed from the container. Thus the number of entries drops with each executed take-operation and the two queries (the read committed query of CAPI-2 and the coordination query of CAPI-3) need to filter a smaller set of entries for subsequent operations. Using one single explicit user transaction, all entries remain in the container until that transaction is committed at the end of the test run and therefore, the number of entries to filter remains the same.

5.2 Memory Utilization

To measure memory utilization a method provided by the .NET Micro Framework is used that forces its garbage collector to run and show the results in the output window.

First the memory utilization was measured before the space was created and used as a basis to calculate the bytes used by the space only. Thereafter, three measurements with a different count of containers and entries have been performed. The entries contain no value, like for example an integer, so only the memory usage of the “infrastructure components” of TinySpaces (entries, containers, selectors, etc.) is measured. Therefore, two write selectors (a FIFO-Selector and Key-Selector) are attached to each entry additionally. The following figure shows the average results of ten test-runs.

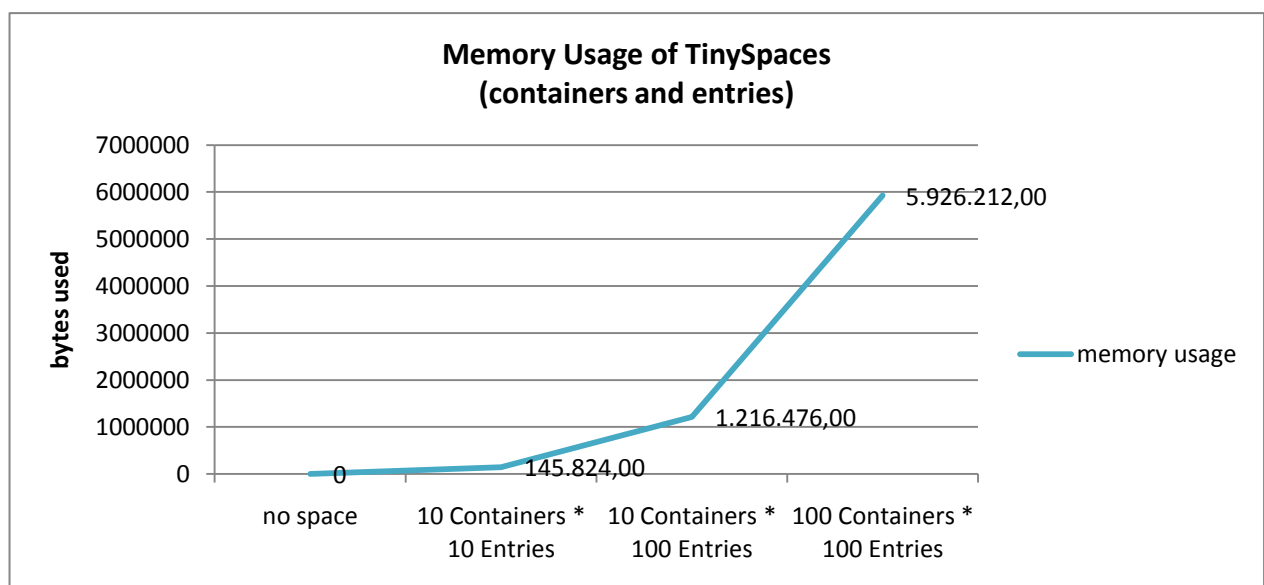


Figure 55 - Memory usage of TinySpaces

The highest amount of entries and containers tested (100 by 100) uses nearly six megabytes of memory. As the smallest device which runs the .NET Micro Framework, which is the Digi Connect ME, has eight megabytes memory available, it could run TinySpaces with a maximum amount of approximately 100 containers with 100 entries each. Therefore at least a space with a dimension of “100 by 100” is manageable by any available .NET Micro Framework device.



Figure 56 - Digi Connect ME⁴

5.3 Power Consumption

Unfortunately the ChipworkX development system does not lower power consumption when its CPU is idle. According to a support administrator of the GHI Electronics Forum, this feature may be added in the next firmware release which includes .NET Micro Framework 4.0 support and is planned for Q1 2010 [56]. As that firmware has not been released yet, this test cannot be performed.

5.4 Serialization Performance

As a fourth test, the three serialization mechanisms provided by the current implementation of TinySpaces (see 4.7.2.2) were benchmarked for performance, code-size and byte usage. For these tests three message types were chosen: “CreateContainerRequest”, “EmptyResponse”, which is used as a kind of “void” response, and “EntryResponse”, which is used as response for read-, and take-operations. The “EntryResponse” was filled with three entries which a 32 bit integer as value. This configuration was used for the following three benchmarks: Serialization performance, deserialization performance and byte usage.

Figure 57 shows the results of the serialization performance benchmark.

⁴ http://www.digi.com/images/products/prd_em_digiconnectme_lg.jpg

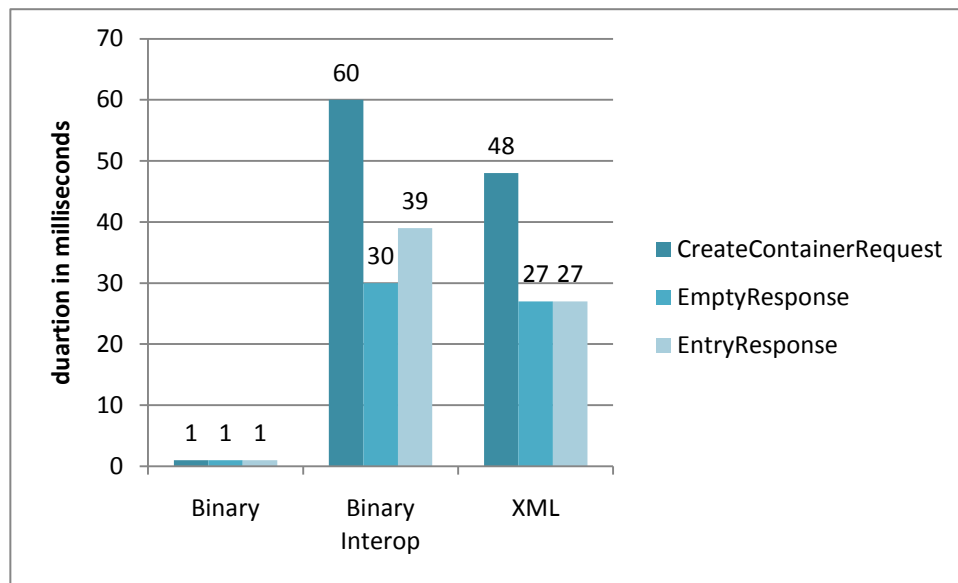


Figure 57 - Serialization performance

As can be seen the built in serialization mechanism of the .NET Micro Framework (*Binary*) totally outperforms the other ones. This was expected as it is implemented in native code (C++) in contrast to the two other ones, which are custom implementations using managed code (C#). Additionally *Binary Interop* is slower than *XML*. That is because *Binary Interop* makes heavy use of Reflection which is performance costly while *XML* requires custom serialization interceptors to be developed for each class that shall be serialized. That, however, affects that *XML* has a bigger code-size than its competitors as will be shown in a subsequent benchmark.

The following figure (Figure 58) shows the deserialization performance of those three mechanisms.

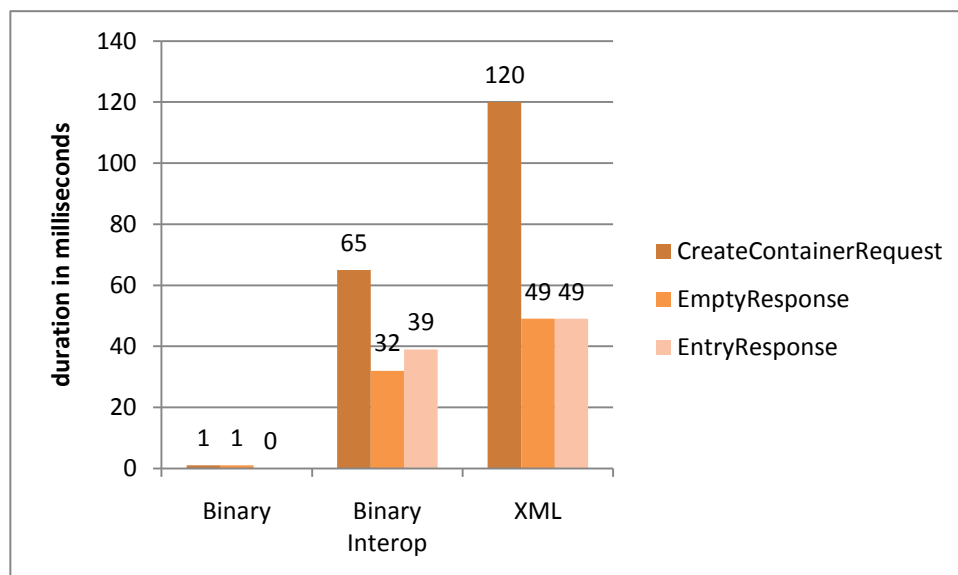


Figure 58 – Deserialization performance

Again the built in serialization mechanism outperforms the other ones as expected. What is interesting here is that *Binary Interop* is now faster than *XML*, which is caused by the fact that the latter needs to parse a comparatively large string in order to de-serialize a message. Thereby, the read values need

to be converted into the appropriate .NET type, like for example the string representation of an integer needs to be converted into a 32 bit integer. *Binary Interop* does not have to do that conversion.

Next the amount of bytes the serialization mechanisms need to store the three message objects has been measured. The results are shown in Figure 59.

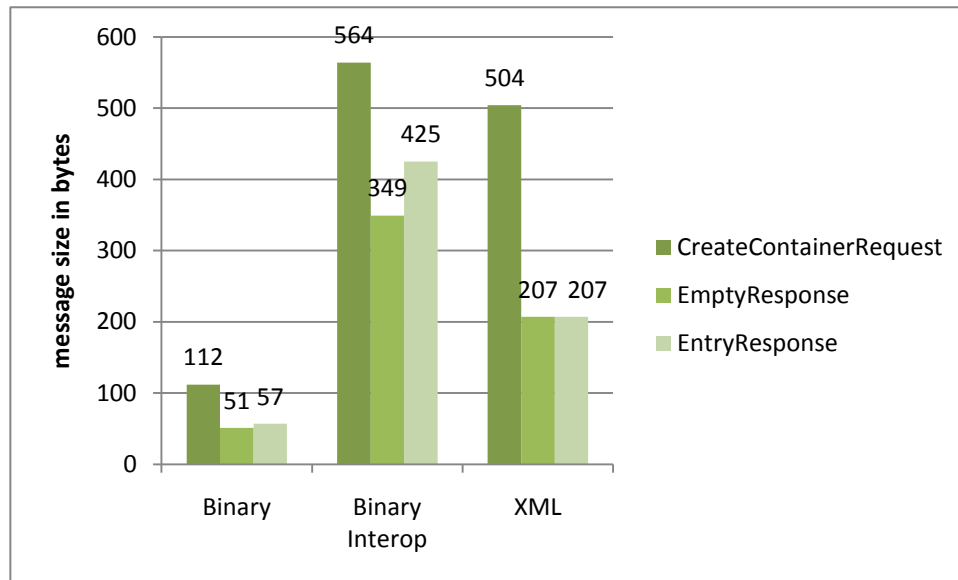


Figure 59 - Message sizes

It does not surprise that *Binary* again outperforms the other formats. What is interesting is that *Binary Interop*, though it is a binary format, creates larger byte arrays than *XML*. The simple reason for this is that the former needs to propagate the type information along with the data. Therefore, the full name of a type which is serialized is stored as a string and thus requires a lot more size than a simple element name with a short prefix as used in *XML*.

Finally Figure 60 shows the code-size of those different serialization mechanisms.

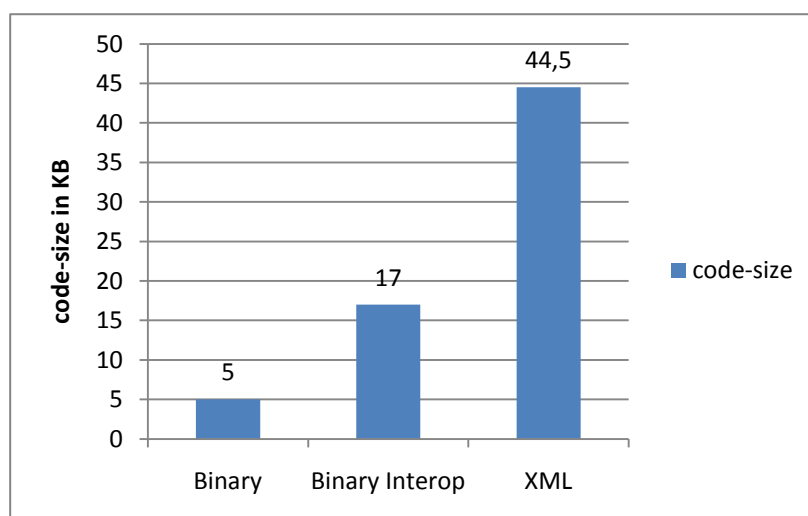


Figure 60 - Code-size

It is clear that the built-in serialization mechanism *Binary* has the smallest code-size, as its logic is built into the .NET Micro Framework itself. The interesting point here is, however, that *Binary Interop* has a

far smaller code-size than *XML*. Moreover the big advantage of *Binary Interop* over *XML* is that it already supports all arbitrary objects which are serializable. In contrast to that *XML* requires an interceptor to be implemented for each class that needs to be serialized, which is then used to serialize and deserialize that object. Therefore, the code-size of *XML* increases with the number of different types that shall be serialized.

As a result each of the serialization mechanisms has its advantages and disadvantages and is applicable for a different scenario:

While *BINARY* offers the best results in all performed benchmarks, it is dependent on the .NET Micro Framework and can only be used to transmit messages between devices which support that platform.

Binary Interop has the worst serialization performance and creates the biggest messages. However, it is capable of exchanging every custom serializable type with all other .NET platforms and does not require custom code to be developed for that purpose. For that reason code-size does not increase with the number of types which have to be serialized.

In contrast to that, *XML* does require a special serialization- and deserialization interceptor to be developed for each custom type that shall be serialized. Thus, code-size increases with the amount of serializable types. However, as it generates XML-messages, this mechanism can be used to communicate with all devices, no matter what platform they are based on.

6 Future Work and Ideas

TinySpaces are the first minimized XVSMP implementation for resource constrained devices and there are still many features which can be added and improved.

- A new XVSMP (Extensible Virtual Shared Memory Protocol) needs to be developed which suits to the new layered architecture of XVSMP and the new features like queries, named containers et cetera.
- Also the new two step approach concerning aspects needs to be reviewed by the XVSMP Technical Board and either rejected or accepted. In the latter case it would need to be built into the new XVSMP.
- Several different communication services could be implemented to leverage Bluetooth, ZigBee, Z-Wave, GSM and other protocols.
- Although a lot of performance improvements have already been made, there is still the need to make TinySpaces faster. During the tests it became clear that a lot of performance is lost within the runtime layer (about 40%) as the logic of the wait handler and the timeout handler is complex. Therefore, this would be good place to start.
- TinySpaces need to be ported to the new .NET Micro Framework as soon as the appropriate firmware is available for the embedded devices. Then the tests need to be re-run, because the main emphasis of the new version of .NET Micro Framework has been put on runtime performance. Consequently the test-results might turn out much better than in this thesis.

Additionally, several additional research subjects have emerged.

- To target even more resource constrained devices, a native implementation in C or C++ may be appropriate. However, this would be a tedious task if it was done manually. For this reason a model driven approach described in [31] should be concerned. All the XVSMP layers could be modeled and then converted into any language. It could then also be used to develop the XVSMP implementations for enterprise systems using Java or .NET with the main advantage that all changes to the model would immediately be reflected in the different technology dependent implementations. This however has to be done with the fact in mind, that the enterprise versions of XVSMP and those for embedded devices, etc. all have different requirements to satisfy. Therefore, it should be possible to compose a specialized XVSMP implementation for a specific scenario.
- If TinySpaces would be ported to other platforms, there might be the need for a binary protocol: XVSMP-Binary. The reason for this is that the mediums those devices use to communicate might have too low data rates to use an XML-based protocol. Furthermore generating and parsing XML files is not efficient.
- As hard real-time constraints are given in many scenarios for embedded systems, research needs to be done to enhance XVSMP with that functionality. How a middleware can be enhanced to support this is explained in [33]. However, this will unfortunately never be supported by TinySpaces as the .NET Micro Framework along with its Tiny CLR is not hard

real-time compatible itself and therefore a native implementation would be required. Additionally, a deal-lock resolving mechanism would be needed as the only way to do this right now is to wait until a request and/or transaction expires, which is obviously insufficient for scenarios with real-time constraints.

- Research has to be done to prove that XVSM is also suitable for coordinating autonomous agents like for example UAVs and UCAVs. However, real-time capabilities are inevitable for this task as can be seen in [35].
- It was shown in [37] that XVSM could be used in ITS. This scenario touches the domain of sensor networks which could be utilized to measure road temperature and humidity and consume that information to send glaze warnings to approaching vehicles, just to give an example. Therefore, a combination of XVSM with TinyDB could be thinkable. However, to be able to send this information in a predictable time to the vehicles, real-time capabilities are a must have. This is another scenario which claims this feature.
- As EMMA, which is introduced in [38], can be used to share sensor information between moving vehicles, it is thinkable, that a space-based approach could be more applicable to achieve this task as those vehicles resemble mobile agents. For that reason TinySpaces need to be ported to a platform which provides better performance and additionally real-time capabilities should be added. It is also thinkable that EMMA and XVSM could be combined to manage coordination of vehicles in ITS.

7 Conclusion

The main focus of this thesis was to show how the model of XVSM can be mapped to an implementation, which satisfies the needs of embedded devices:

- Low code-size
- Low power consumption
- High runtime performance (CPU and memory usage)

It was shown how the different layers of the layered architecture of XVSM were implemented to build TinySpaces. Today it is the first object-oriented implementation of XVSM which follows the layered architecture defined in the formal model.

Moreover it became clear, that some parts of the formal model of XVSM needed to be omitted, like CAPI-3 support of the isolation level Read Committed for the coordinator's accountant information, and some rules had to be violated, like the fact that aspects obtain a reference to the XVSM API, to obtain a version of XVSM which is applicable for resource-constrained devices.

A special emphasis has been put on networked embedded devices. Thus different serialization mechanisms have been introduced and compared to each other in terms of speed, code-size, platform independence and bandwidth usage. Each suits best in a particular scenario.

Different benchmarks have been made to determine how TinySpaces perform on embedded devices. Although no comparable middleware exists, which could be taken as reference, the results give an idea of the capabilities and the limits of TinySpaces. Power consumption could not be measured as the device used for these tests did not support this feature at that time.

The conclusion is, that it is possible to develop a hardware independent XVSM based middleware for embedded devices using the .NET Micro Framework, which mostly complies with the XVSM specification and still offers adequate runtime performance and code-size to be used in small scenarios and even on the smallest existing device, which is the Digi Connect ME with a 55 Mhz CPU and 8 megabytes memory. As the current implementation of TinySpaces can be used to rapidly prototype solutions and test scenarios, it needs to be implemented in native code in order to increase its overall performance and make it applicable for scenarios where a higher throughput of requests is required.

Abbreviations

ACID	A tomicity- C onsistency- I solation- D urability
API	A pplication P rogramming I nterface
CAPI	C ore A pplication P rogramming I nterface
CORBA	C ommon O bject R equest B roker A rchitecture
ECMA	E uropean C omputer M anufacturers A ssociation
EMMA	E mbedded M iddleware in M obility A pplications
FIFO	F irst-In F irst-Out
HAL	H ardware A bstraction L ayer
IL	I ntermediate L anguage
LIFO	L ast-In F irst-Out
MDA	M odel D riven A rchitecture
MDD	M odel D riven D evelopment
MEDC	M obile and E mbedded D evelopers C onference
OS	O perating S ystem
PAL	P latform A bstraction L ayer
P2P	P eer-to- P eer
QoS	Q uality of S ervice
RPC	R emote P rocedure C alls
SoC	S eparation of C oncerns
SPOT	S mart P ersonal O bject T echnology
SQL	S tandard Q uery L anguage
TB	X VSM T echnical B oard
UACV	U nmanned C ombat A ir V ehicles
UAV	U nmanned A ir V ehicles

UID	U nique I dentifier
UML	U nified M odeling L anguage
XVSM	e X tensible V irtual S hared M emory
XVSMP	e X tensible V irtual S hared M emory P rotocol

Table of Figures

Figure 1 - Architecture of the .Net Micro Framework [6]	10
Figure 2 - RPC Communication.....	12
Figure 3 - Message Passing	13
Figure 4 - Message Queuing	13
Figure 5 - Communication using a space [7 S. 11]	14
Figure 6 – Producer/observer implementation with JavaSpaces	15
Figure 7 – Producer/observer implementation with XVSM	16
Figure 8 - Layered Architecture of XVSM.....	17
Figure 9 - Destroying an entry from a container with multiple coordinators assigned	20
Figure 10 – FIFO-Selector	20
Figure 11 – LINDA-Selector	21
Figure 12 - XVSM Runtime.....	23
Figure 13 - USBizi by GHI ElectronicsError! Hyperlink reference not valid.	25
Figure 14 - Layered Architecture of .NET Micro Framework [6].....	29
Figure 15 - Related Work Graph	33
Figure 17 - Direct dependency between components.....	34
Figure 18 - Component interfacing contract	35
Figure 19 - Replacing a component using Contract First Design.....	35
Figure 20 - Contracts of CAPI-1	36
Figure 21 - CAPI-1: Concurrent operations.....	39
Figure 22- Contracts of CAPI-2	42
Figure 23- Using ILockable in CAPI-2	48
Figure 24 - Reading from a container with a transaction	49
Figure 25 - Transaction in inconsistent state.....	51
Figure 26 – Commit/Rollback Information Provider Interfaces.....	51
Figure 27 - Implicit transactions and performance	53
Figure 28 - CAPI-3 Contracts	54
Figure 29 - CAPI-3: Violating the layered Architecture	54

Figure 30 - CAPI-3: Relation of IXQuery and Selector	57
Figure 31 - FIFO-query versus Cnt-query	58
Figure 32 - CAPI-3: Creating queries	61
Figure 33 - CAPI-3: Synchronization of Coordinators	62
Figure 34 - CAPI-4: Contracts	64
Figure 35 - Aspects accessing the API	67
Figure 36 - Using a 16 bit flag enumeration for SpaceIPoint	67
Figure 37 - Space Aspect Registry.....	70
Figure 38 - Notification process.....	71
Figure 39 - Notification implementation of TinySpaces	72
Figure 40 - Adding an aspect implemented in JAVA using XVSMP	73
Figure 41 - Adding an aspect implemented in Microsoft .NET using XVSMP	73
Figure 42 - Adding an aspect in general using the technology independent approach	74
Figure 43 - CAPI-5: Contracts	75
Figure 44 - Deadlock caused by aspect with single-threaded CoreProcessor	76
Figure 45 - TinySpace Runtime.....	76
Figure 46 - CoreProcessor	77
Figure 47 - RequestMessageContent	78
Figure 48 - WaitHandler	79
Figure 49 - Request expires erroneously	81
Figure 50 - Deadlock with transactions	83
Figure 51 - Example scenario with multiple transport protocols.....	84
Figure 52 - CAPI-5b: Contracts	85
Figure 53 - ChipworkX Development Sastem V1.2 (left) and ChipworkX Module (right).....	88
Figure 54 - Write benchmark	89
Figure 55 - Take benchmark	90
Figure 56 - Memory usage of TinySpaces	91
Figure 57 - Digi Connect ME	92
Figure 58 - Serialization performance	93
Figure 59 – Deserialization performance	93

Figure 60 - Message sizes	94
Figure 61 - Code-size	94

References

1. **Peng, J. and Liao, J.** Design and Performance Evaluation of emORB for Pervasive Computing. *Pervasive Computing and Applications*. 2008, Vol. 1, pp. 185-189.
2. **Kühner, Jens.** *Expert .Net Micro Framework*. s.l. : Apress, 2008.
3. **Noergaard, Tammy.** *Embedded Systems Middleware: Understanding File Systems, Databases, Virtual Machines, Networking and More!* s.l. : Butterworth Heinemann, 2008.
4. Microsoft Launches Smart Personal Object Technology Initiative. *Microsoft PressPass*. [Online] November 17, 2002. [Cited: December 7, 2009.] <http://www.microsoft.com/presspass/features/2002/nov02/11-17SPOT.mspx>.
5. **Thompson, Donald and Miller, Colin.** *.Net Micro Framework White Paper*. s.l. : Microsoft, 2007. pp. 1-23.
6. **Microsoft.** Understanding the .NET Micro Framework Architecture. *.NET Micro Framework Porting Kit Help*. s.l. : Microsoft, 2009.
7. **Scheller, Thomas.** *Design and Implementation of XcoSpaces, the .Net Reference Implementation of XVSM - Core Architecture and Aspects*. Vienna : Master Thesis, Insutute of Computer Languages, 2008.
8. **Schreiber, Christian.** *Design and Implementation of MozartSpaces, the Java Reference Implementation of XVSM - Custom Coordinators, Transactions and the XVSM protocol*. Vienna : Master Thesis, Insutute of Computer Languages, 2008.
9. **Pröstler, Michael.** *Design and Implementation of MozartSpaces, the Java Reference Implementation of XVSM - Timeout Handling, Notifications and Aspects*. Vienna : Master Thesis, Insutute of Computer Languages, 2008.
10. **Karolus, M.** *Design and Implementation of XcoSpaces, the .Net Reference Implementation of XVSM - Coordination, Transactions and Communication*. Vienna : Master Thesis, Insutute of Computer Languages, 2010.
11. **Kühn, eva, Craß, Stefan and Salzer, Gernot.** *Algebraic Foundation of a Data Model for an Extensible Space-Based Collaboration Protocol*. Vienna : s.n., 2009.
12. **Craß, Stefan.** *A Formal Model of the Extensible Virtual Shared Memory (XVSM) and its Implementation in Haskell*. Vienna : Master Thesis, Insutute of Computer Languages, 2010.
13. SpaceBasedComputing - Home. *SpaceBasedComputing*. [Online] 2007. [Cited: December 07, 2009.] <http://www.spacebasedcomputing.org/home.html>.
14. **White, James.** *RFC# 707 - A High-Level Framework for Network-Based Resource Sharing*. 1976.

15. **Mamoud, Q. H.** Sun Developer Network (SDN). *Getting Started With JavaSpaces Technology: Beyond Conventional Distributed Programming Paradigms*. [Online] Sun, 07 12, 2005. [Cited: 02 03, 2010.] <http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/>.
16. **Tuwis++.** *Verteiltes Programmieren mit Space Based Computing Middleware*. [Online] Vienna University of Technology, Institute of Computer Languages, 2010. [Cited: 02 04, 2010.] http://tuwis.tuwien.ac.at/zope/_ZopelId/04743907A4Pio5uL78w/tpp/lv/lva_html?num=185226&sem=2010S.
17. **Fensel, D., et al.** *Queues Are Spaces - Yet Still Both Are Not The Same?* s.l. : Technical report, 2007. url: http://www.spacebasedcomputing.org/fileadmin/files/SBC-QueuesAreSpaces_20070511.pdf.
18. *Introducing the concept of customizable structured spaces for agent coordination in the production automation domain.* **Kühn, E., et al.** Vienna : International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, 2009. Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems. pp. 625-632. 978-0-9817381-6-1.
19. **Richards, Mark.** *Java Transaction Design Strategies*. s.l. : C4Media, 2006.
20. **Marek, Alexander.** *Profile for XcoSpaces: AdvancedPersistency and TupleConverter, Praktikum*. Vienna, Vienna, Austria : Vienna University of Technology, Institute of Computer Languages, E185/1 : Space Based Computing Group, 7 2008.
21. **Marwedel, Peter.** *Embedded System Design*. 1st. Berlin : Springer, 2003.
22. **Zurawski, Richard.** *Embedded Systems Handbook*. s.l. : CRC Press, 2006.
23. **J.-M. Bergé, O. Levia, J. Rouillard.** *High-Level System Modeling*. s.l. : Kluwer Academic Publishers, 1995.
24. **Kopetz, H.** *Real-Time Systems - Design Principles for Distributed Embedded Applications*. s.l. : Kluwer Academic Publishers, 1997.
25. **Microsoft.** *MICROSOFT .NET MICRO FRAMEWORK VERSION 4.0*. s.l. : Microsoft, 2009.
26. MSDN. *Garbage Collection*. [Online] Microsoft, 11 2007. [Cited: 02 03, 2010.] <http://msdn.microsoft.com/de-de/library/0xy59wtx.aspx>.
27. **Microsoft.** Home Controls Manufacturer Uses .NET Micro Framework to Create Product Quickly. *Microsoft Case Studies*. [Online] 8 5, 2008. [Cited: 12 16, 2009.] http://www.microsoft.com/casestudies/Case_Study_Detail.aspx?CaseStudyID=201472.
28. **Hanselman, Scott and Miller, Colin.** Hanselminutes on 9 - The .NET Micro Framework with Colin Miller. *channel9*. [Online] [Cited: 12 16, 2009.] <http://channel9.msdn.com/posts/Glucose/Hanselminutes-on-9-The-NET-Micro-Framework-with-Colin-Miller/>.

29. Driver Safety Device Uses .NET Micro Framework. *.NET Micro Framework*. [Online] 2009. [Cited: 12 16, 2009.] <http://download.microsoft.com/download/4/C/6/4C66CF39-348D-4B4A-AC30-0B70B0D2B863/inthinc%20Waysmart%20Case%20Study.pdf>.
30. OMG. *OMG Model Driven Architecture*. [Online] OMG, 08 12, 2009. [Cited: 02 03, 2010.] <http://www.omg.org/mda/>.
31. *A model-driven design environment for embedded systems*. **Riccobene, E., et al.** New York, NY, USA : ACM, 2006. Proceedings of the 43rd annual Design Automation Conference. pp. 915 - 918. 1-59593-381-6.
32. *Model-integrated development of embedded software*. **Karsai, G., et al.** Nashville, TN, USA : IEEE, 2003. Proceedings of the IEEE. Vol. 91, pp. 145- 164.
33. *An Overview of the Real-time CORBA Specification*. **Schmidt, D. C. and Kuhns, F.** Special Issue on Object-Oriented Real-time Distributed Computing, s.l. : IEEE Computer, 2000, Vol. 33, pp. 56-63. 6.
34. **Schmidt, D. C., et al.** A High-Performance Endsystem Architecture for Real-time CORBA. *IEEE Communications Magazine*. 2, 1997, Vol. 14, Distributed Object Computing.
35. *The OCP - An Open Middleware Solution for Embedded Systems*. **Paunicka, J., Mendel, B. and Corman, D.** Arlington : s.n., 2001. Proceedings of the American Control Conference. pp. 3345-3350.
36. *Analysis on Open Control Platform*. **Yin, Y., Wang, Y. and Wang, Y. X.** 2007. IEEE International Conference on Mechatronics and Automation. pp. 3371 – 3376.
37. **Kühn, E., et al.** Aspect-Oriented Space Containers for Efficient Publish/Subscribe Scenarios in Intelligent Transportation Systems. *Lecture Notes in Computer Science*. 2009, Vol. 5870/2009, pp. 432-448.
38. **Katramados, I., et al.** Heterogeneous sensor integration for intelligent transport systems. *Road Transport Information and Control - RTIC 2008*. May 20-22, 2008, pp. 1-8.
39. **Levis, P., et al.** TinyOS: An Operating System for Sensor Networks. *Ambient Intelligence*. s.l. : Springer, 2004.
40. *The nesC Language: A Holistic Approach to Networked Embedded Systems*. **Gay, D., et al.** New York, NY, USA : ACM, 2003. Programming Language Design and Implementation (PLDI). 1-58113-662-5.
41. *TinyGALS: A Programming model for event-driven embedded systems*. **Cheong, E., et al.** New York, NY, USA : ACM, 2003. 2003 ACM Symposium on Applied Computing. pp. 698 - 704. 1-58113-624-2.
42. *TinyDB: an acquisitional query processing system for sensor networks*. **Madden, S.R., Franklin, M.J. and Hellerstein, J.M.** New York, NY, USA : ACM, 2005. Vol. 30, pp. 122-173. 0362-5915.
43. *An Embedded Middleware Platform for Pervasive and Immersive Environments for-All*. **Baldoni, R., et al.** s.l. : IEEE, 2009. Sensor, Mesh and Ad Hoc Communications and Networks Workshops, 2009. SECON Workshops apos;09. 6th Annual IEEE Communications Society Conference. pp. 1-3.

44. *The many faces of publish/subscribe*. **Eugster, P. Th., et al.** ACM Computing Surveys (CSUR), New York, NY, USA : ACM, 2003, Vol. 35, pp. 114 - 131.
45. SpringSource.org. *SPRING source*. [Online] 2010. [Cited: 02 04, 2010.] <http://www.springsource.org/>.
46. *Component-based approach for embedded systems*. **Crnkovic, I.** Oslo, Norway : s.n., 2004. 9th International Workshop on Component-Oriented Programming.
47. MSDN. *Delegates*. [Online] Microsoft, 11 2007. [Cited: 02 03, 2010.] <http://msdn.microsoft.com/de-de/library/system.delegate.aspx>.
48. MSDN. *System.Reflection Namespace*. [Online] Microsoft. [Cited: 02 03, 2010.] <http://msdn.microsoft.com/en-us/library/cc544889.aspx>.
49. MSDN. *Generics in the .NET Framework*. [Online] Microsoft. [Cited: 02 03, 2010.] <http://msdn.microsoft.com/en-us/library/ms172192.aspx>.
50. **Bracha, G.** Java Sun. *Generics Tutorial*. [Online] 07 05, 2004. [Cited: 02 03, 2010.] <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>.
51. **ISO**. ISO/IEC 9075:1992, Database Language SQL. s.l. : International Organization for Standardization, 1992.
52. **Craß, Stefan**. *Notes on XVSM Transactions*. Vienna : Vienna University of Technology, Insutute of Computer Languages, 2009.
53. MSDN. *Object-Oriented Programming (C# and Visual Basic)*. [Online] Microsoft. [Cited: 02 03, 2010.] [http://msdn.microsoft.com/en-us/library/dd460654\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd460654(VS.100).aspx).
54. ZigBee Alliance. *ZigBee*. [Online] ZigBee Alliance. [Cited: 02 03, 2010.] <http://www.zigbee.org/>.
55. GHI Electronics. *ChipworkX Development System*. [Online] GHI Electronics. [Cited: 02 03, 2010.] <http://www.ghielectronics.com/product/125>.
56. GHI Electronics Forum. *Measure cpu & ram usagg and power consumption*. [Online] 01 01, 2010. [Cited: 01 02, 2010.] <http://www.ghielectronics.com/forum/index.php/topic,2704.0.html>.