FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Implementation of XVSM for the iOS platform

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Gerald Grötz
Matrikelnummer 0427554

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn
Mitwirkung: Dipl.-Ing. Tobias Dönz
Dipl.-Ing. Stefan Craß

Wien, 29.08.2013

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Implementation of XVSM for the iOS platform

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Medical Informatics

by

Gerald Grötz
Registration Number 0427554

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn
Assistance: Dipl.-Ing. Tobias Dönz
Dipl.-Ing. Stefan Craß

Vienna, 29.08.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Gerald Grötz
Dr. Czermakstraße 15/1, 2000 Stockerau

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 Verfasserin)

Danksagung

Meine Studien und im speziellen diese Arbeit wäre ohne die Mithilfe und Unterstützung vieler Personen nicht möglich gewesen und deshalb möchte ich mich bei Ihnen recht herzlich bedanken.

Besonders erwähnen möchte ich an dieser Stelle meine Eltern Helmut und Maria die mich nicht nur finanziell unterstützt, sondern auch mit Geduld und Verständnis meinen Weg begleitet haben. Ihnen habe ich zu verdanken, dass ich nicht früher ohne Abschluss das Studium beendet habe. Ebenso möchte ich mich bei meinen zwei Brüdern Harald und Andreas sowie meiner Schwägerin Karin für gelegentliche fachliche Ratschläge beziehungsweise die aufmunternden Worte bedanken, die mich zum Weitermachen animierten. Die Gespräche mit meinem Großvater Josef halfen mir auch, immer neue Sichtweisen zu entwickeln. Leider kann sich meine Großmutter Aloisia nicht mehr mit mir gemeinsam über den Abschluss dieser Arbeit freuen. Ebenfalls möchte ich an dieser Stelle die Unterstützung durch meine Freunde - insbesondere Kirchmauer Alexander und Reiff Christian - erwähnen.

Ein Höhepunkt meines Studium war sicherlich das Auslandssemester in Göteborg, bei dem ich mich persönlich weiterentwickeln konnte und das mir sprachlich sehr geholfen hat.

Schlussendlich möchte ich mich bei meinen Betreuern eva Kühn, Tobias Dönz und Stefan Craß für die Mithilfe an dieser Arbeit bedanken.

Abstract

Due to the increasing complexity of software systems there comes the need of technologies that help developers to simplify the programming process. Middleware systems in general can offer this functionality. One type of middleware is based on the Space Based Computing paradigm. It offers a shared memory data space that can be accessed concurrently by different users. The eXtensible Virtual Shared Memory (XVSM) architecture uses this approach and provides an easy extendable solution for developers. Actual implementations are based on Java (MozartSpaces) and .NET (XCOSpaces). With the rapid growth of smartphones based on Apple's iOS operating system comes the need of an implementation for that platform.

The goal of this thesis is the provision of an implementation for the iOS platform that is fully compatible to MozartSpaces, the actual reference implementation of XVSM. A research process concerning possible solutions is followed by a ready to use implementation for software developers.

The output is evaluated by different kind of performance benchmarks. The compatibility to MozartSpaces is evaluated by integration tests and presented by an application scenario where the new implementation works hand in hand with Mozartspaces.

The new implementation is based on the native programming language of Apple, Objective C and works well in association with MozartSpaces.

Kurzfassung

Aufgrund der zunehmenden Komplexität von Softwaresystemen ist die Verwendung von Technologien notwendig, die den Entwicklern die Programmierung vereinfachen. Middleware Systeme sind eine Möglichkeit das zu erreichen. Ein Typ von Middleware basiert auf der Space Based Computing Paradigma. Sie bietet einen gemeinsamen Datenraum (Space), der gleichzeitig von verschiedenen Benutzern verwendet werden kann. Die eXtensible Virtuelle Shared Memory (XVSM) Architektur nutzt diesen Ansatz und bietet eine einfache erweiterbare Technologie für Entwickler. Aktuelle Implementierungen basieren auf Java (MozartSpaces) und .NET (XCOSpaces). Mit dem rasanten Wachstum von Smartphones basierend auf Apple's iOS Betriebssystem kommt die Notwendigkeit einer Implementierung für diese Plattform.

Das Ziel dieser Diplomarbeit ist die Bereitstellung einer Implementierung für die iOS-Plattform die vollständig kompatibel zu MozartSpaces, der aktuellen Referenz-Implementierung von XVSM ist. Am Beginn steht der Vergleich möglicher Lösungsansätze mit anschließender Implementierung für diese Plattform.

Die Implementierung wird durch verschiedene Arten von Performance-Benchmarks ausgewertet. Die Kompatibilität zu MozartSpaces wird durch Integration-Tests evaluiert und ein Anwendungsszenario präsentiert die Zusammenarbeit zwischen der neuen Implementierung und MozartSpaces.

Die neue Implementierung basiert auf Objective C, der nativen Programmiersprache von Apple und funktioniert gut in Verbindung mit MozartSpaces.

Contents

List of Listings

List of Figures

List of Tables

List of Abbreviations

AOT	Ahead-Of-Time
API	Application Programming Interface
ARC	Automatic Reference Counting
CAPI	Core API
GC	Garbage Collection
GCD	Grand Central Dispatch
GUI	Graphical User Interface
IDE	Integrated Development Environment
IP	Internet Protocl
JIT	Just-in-time
JVM	Java Virtual Machine
LTS	Lime Tuple Space
NAT	Network Address Translation
QoS	Quality of Service
RPC	Remote Procedure Call
SDK	Standard Development Kit
STUN	Session Traversal Utilities for NAT
URL	Uniform Resource Locator
WSN	Wireless Sensor Networks
XML	eXtensible Markup Language
XVSM	eXtensible Virtual Shared Memory
XVSMP	XVSM Protocol
XVSMQL	XVSM Query Language

CHAPTER 1

Introduction

Due to the falling prices of smartphones, mobile computing and mobile Internet access gain more and more influence in our everyday work and life. Figure 1.1 with data from [2] show the annual turnover (2010 and 2011) and predicted turnovers (2012 to 2016) of smartphones worldwide. Figure 1.2 shows the worldwide proportion of smartphone operating systems in the first quarter 2012 ([18]) and the prediction of 2015 ([1]). With this trend in mind there exists the possibility of building complex and large applications because great sales can be expected. For example, map apps, collaborating games or social media network access are possible fields for development. There exist mobile applications in the application store of Apple [4] and Google [33] that are larger than 100 megabyte. In addition to the size of the application they are inherently distributed applications. Due to such a dramatically increase of complexity of smartphone applications there comes the need of a technology that can assist designers as well as developers to quickly build reliable and reusable software. Like on desktop computers a middleware should achieve that. This should also allow communication between regular personal computers and smartphones. For the perspective of the application there should be no difference between these devices.

One kind of middleware paradigm to achieve these requirements called Space Based Computing uses some kind of blackboard (called space) to share data between hosts. It offers the possibility to exchange data no matter the receiving endpoint is operating or not (time decoupling) as well as it is not necessary to know the address of the receiver (space decoupling). An adaptable and extendable implementation of this paradigm is XVSM.

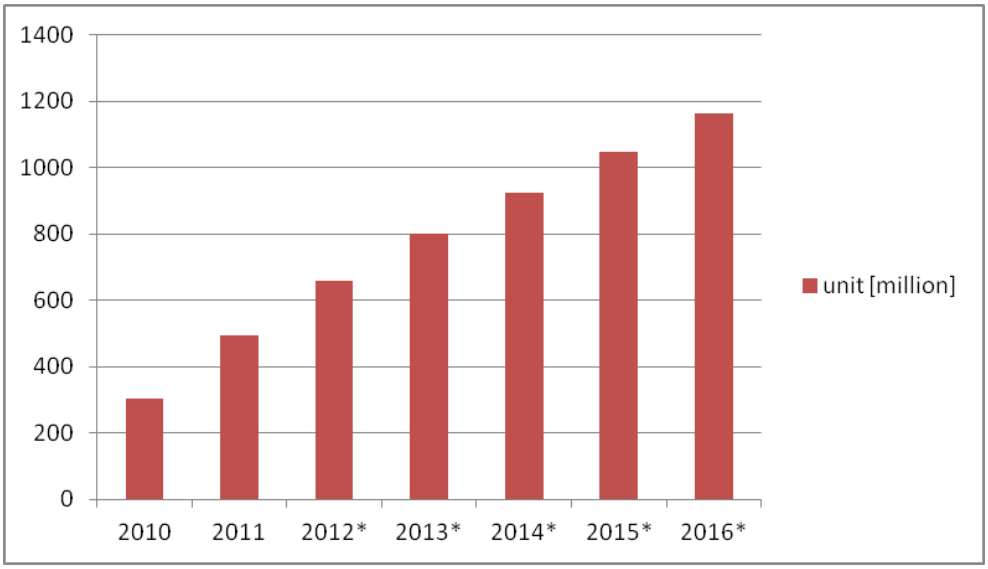

Figure 1.1: Smartphone turnover 2010 and 2012; prediction from 2012 to 2016

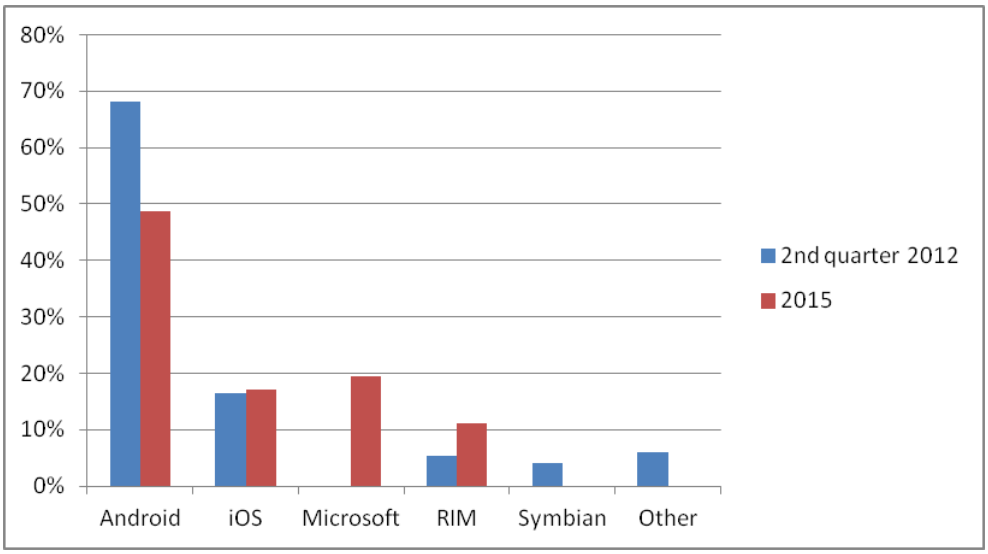

Figure 1.2: Proportion of smartphone operating systems

1.1 Motivation and goals

Since hardware capabilities of actual mobile devices improved rapidly in the last years there are new possibilities for designing mobile applications. Processing power of multicore CPUs and clock speed beyond 1GHz offer the possibility of developing complex applications. In addition, wireless communication between the devices using 3G/4G and WLAN allows high data traffic. Applications in AppStores of Apple and Google exceed the 100MB boundary and are often distributed applications. Due to these facts there comes the need of software systems that reduce complexity for developers. The Space Based Computing paradigm based on the Linda coordination model, e.g. XVSM offers a solution by decoupling communication in space and time. This approach works well for static as well as for dynamic clients and is therefore well suited for mobile devices as mentioned in [CFL+06]:

> "Summarizing, the Linda coordination model grants the flexibility and the adaptability needed in developing applications in mobile computing scenarios."

The main goal of this thesis is a XVSM implementation for the iOS platform (iPhone, iPad, iPod) that is fully compatible with the actual MozartSpaces version. This includes the following parts:

- Operations, coordination and transactions
- Runtime model, communication protocol and API semantics
- Semantics of aspects
- Persistency operations

Different possible solutions should be mentioned to achieve this goal like using the existing source code and adapt it for using on iOS or a new implementation. Since Apple included a lot of restrictions in the API of iOS in contrast to OS X as well as in the distribution mechanism of Apps there may be some unexpected limitations in the result of this thesis. Some solutions will be presented to outwit these limitations including mentioning the consequences. Additionally, the restrictions by the user interface for configuration and running the middleware like no command line interface and limited interaction possibilities (e.g. no physical keyboard or limited screen size) must be handled by designing an intelligent user interface. Another challenge is the design of a serialization mechanism that operates in the heterogeneous environment.

A small scenario where the MozartSpaces implementation as well as the new iOS implementation works hand in hand will be created. This will be demonstrated by a small example that presents the opportunities of XVSM on the iOS platform interacting with the existing MozartSpaces implementation.

To achieve compatibility the actual integration test suite of MozartSpaces will be executed by using a remote core of the new Objective C implementation. In addition, some benchmarks will be performed to compare the different implementations concerning execution speed and memory consumption.

1.2 The thesis' structure

This thesis is structured in the following chapters: Chapter 2 is about middleware in general, XVSM including the formal definition, aspects about using mobile devices and in particular iOS devices, requirements for porting MozartSpaces and communication in heterogeneous systems. Chapter 3 describes the related work and why a new implementation for the iOS platform is necessary. Chapter 4 is about possible use cases of the new Objective C implementation for the iOS platform. Chapter 5 sums up the most important aspects of the implementation details. This includes how the porting process is done, some issues concerning iOS and detailed information about the implementation. Chapter 6 describes an application scenario that present the new Objective C implementation and MozartSpaces working hand in hand. Chapter 7 describes the evaluation concerning execution performance, scalability and compatibility to MozartSpaces. The deploying process on iOS devices including some Apple specific issues is outlined in chapter 8. Chapter 9 describes open issues concerning the new Objective C implementation. Finally, chapter 10 summarizes the thesis.

CHAPTER 2

Background

To get an understanding of the content of the thesis some concepts needs to be described. This chapter gives an instruction of middleware in general (see 2.1) including a short description of Space Based Computing. Further, an introduction to XVSM (see 2.2) is given with the description of the formal definition, which it is based on. Afterwards, issues concerning mobile devices are discussed (see 2.3). Then, aspects about the iOS platform are mentioned (see 2.4). The chapter gets finished with an overview about the porting MozartSpaces (see 2.5) and some concerns about communication in heterogeneous systems (see 2.6).

2.1 Middleware

Due to the increasing size and complexity of software systems there comes the need of hiding details from developers to simplify the programming process. Middleware systems offer such functionality. They hide complexity of distributed systems, e.g. concurrency access and give a well-defined view on the system.

The first part of this chapter describes general aspects of middleware. The second part is about Space Based Computing at a glance. After that some aspects of middleware and mobile devices are given.

2.1.1 General aspects

Definition

[CDK05, p. 16] gives a overview of middleware and describes it in useable way:

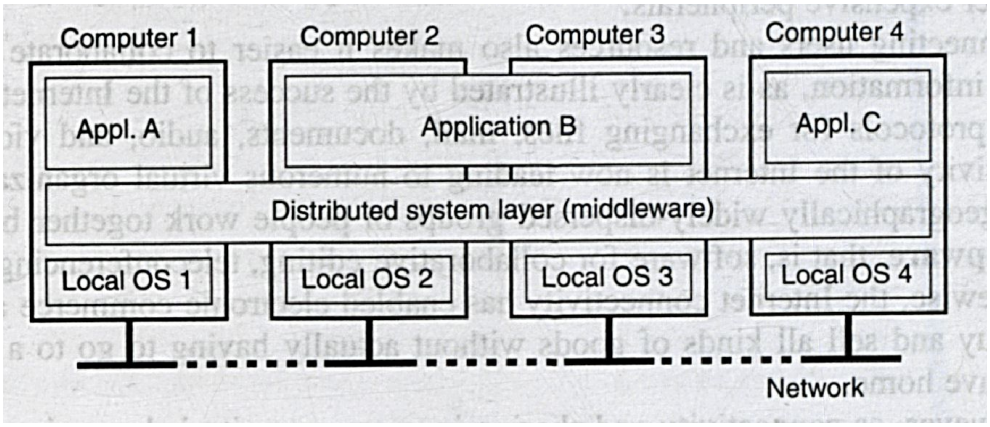

Figure 2.1: A distributed system organized as middleware ([TS06])

> "The term middleware applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages."

Middleware provides a software layer (2.1) that is between operating system and the application. It offers a higher degree of abstraction and is therefore easier to handle. Therefore the view of the application (API, programming concept) is the same as it would be on a single computer system although it is a distributed system. A middleware offers the ability to focus on the application logic but on low-level details like concurrency control or transaction management. There are 7 challenges defined in [CDK05] that need to be fulfilled for distributed systems. A middleware system takes over parts of them and hides them from the developer:

- **Heterogeneity**: Various kinds of systems (e.g. networks, operating systems, programming languages) should work together using well-defined interfaces and protocols (e.g. IP, XML). This allows overcoming differences like unequal data type representation or usage of different programming languages.
- **Openness**: Interfaces should be published, accessible and standardized to allow extension or reimplementation.
- **Security**: Since important data is transmitted it must be secured, especially based on confidentiality (protection against intruders), integrity (protection against modification) and availability (protection against interruption).
- **Scalability**: The system should be extendable with resources as well as with users and still work efficiently.

- **Failure handling**: The systems should be able to detect hardware and software failures and handle them accordingly. For instance retransmitting of corrupted data or recovering after an error.
- **Concurrency**: The system should handle data access to resources at the same time and still stay in a consistent state.
- **Transparency**: "Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components [CDK05, 23]." The most important transparencies are the following:
  - **Access**: Local and remote access use the same operations.
  - **Location**: The physical or network location is hidden.
  - **Concurrency**: Allow different processes accessing shared resources at the same time and keeping them consistent.
  - **Replication**: Resources may be replicated for performance, availability or security reasons without reflecting it to users or programmers.
  - **Failure**: Failure and recovery are hidden for the resource.
  - **Mobility**: Allows the movement of resources without reflecting it to the user.
  - **Performance**: The system can be reconfigured to handle different loads.
  - **Scaling**: The systems need no changes when extending it.

Categories of middleware

[JYYL09] describes four different categories of middleware systems:

- **Procedural middleware** allows "Remote Procedure Calls" (RPC) to communicate between the peers. A server provides different procedures that can be executed by the client. An advantage is the simple way of implementation and the good support by the operating systems. Disadvantages are for example the lack of support for replication, the non-existence of asynchronous communication or a lack of scalability. An implementation is for instance the "CORBA".
- **Message-oriented middleware (MOM)**: The communication is done by exchanging messages. There exist at least two different types of MOM: message queuing and publish-subscribe. In the first type of MOM, the messages are sent to a queue and from there forwarded to the receiver (indirect). In the second case the messages are sent to interested parties that have been registered before. An advantage

is the fact that group communication and synchronous/asynchronous communication are supported. A disadvantage is the need of an extra component (message broker). An implementation is for instance the "Java Message Service" (JMS).

- **Transactional middleware**: The communication supports distributed transaction according to the ACID ([HR83]) properties. An advantage is that the system is always in a consistent state. A disadvantage is the sometimes undesired-overhead. An implementation is for instance "Customer Information Control System" (CICS).

- **Object middleware**: This is an extension to the procedural approach. Some object-oriented features like inheritance, object references and exceptions are added to the RPC principle. An advantage is the flexibility - it can replace the other approaches. A disadvantage is the rather heavyweight design. An implementation is for instance "Component Object Model" (COM)

Benefits of using middleware

Due to the requirement of building software in time and budget some new approaches need to be invented. The systematic reuse of artifacts as described in [SB03] is one way to achieve this. There are three main improvements for developing and evolving of application software mentioned which are reached due to middleware:

- **Open standards**: Offers the opportunity of interoperable artifacts with respect to security, layered distributed resource management and fault tolerant services.

- **Strategic focus**: Allows the developers to focus on higher level software artifacts like business logic instead of low level operating system concerns.

- **Implementation reuse**: The additional effort of developing amortizes after reuse.

In [SS02] the benefits of well-designed middleware systems are summarized as follows:

- Hiding error-prone details like socket programming from application developers.

- Costs can be reduced due to reusing artifacts and using previous development expertise.

- Offers a higher-level abstraction to the developers.

- Offers proven services like logging and security for the developer.

2.1.2 Space Based Computing

The Space Based Computing paradigm is based on the Linda Tuple Space communication model by David Gelernter ([Gel85]). It provides a repository of tuples (space) that can be accessed by different processes. Instead of sending messages or using remote procedure calls data exchange and coordination between peers is accomplished by using a shared space. That means that there is no direct communication between the processes. A timeout specifies how the operations on the space are handled - the operation blocks until the timeout expires (from zero seconds to infinity). A space is sometimes understood as a blackboard. It is a shared data space that can be accessed by different atomic operations:

- read (rd): Search the space for an entry (e.g. using template matching, unique id) and return the result.
- write (out): Write a tuple/entry into the space.
- take (in): It is like the read operation except that the entry/tuple is removed from the space (consuming read).

The communication is decoupled in time, space and reference. Due to these characteristics it is intended for the use in large distributed systems [Mor10].

There exist many implementations with different goals on different platforms. This thesis is about the implementation of a technology called XVSM that is based on the space based computing paradigm.

2.2 XVSM

Many program designs are based on concepts like stacks or queues. This also applies in distributed environments. Since template matching does not offer a sufficient way to coordinate processes with respect to this requirement there was a need of an extension. Another need was platform independency to serve the need of the actual internetworking. XVSM [49], developed by the Space Based Computing group [48] was introduced to extend the potential of such a Space Based Computing approach and add the two mentioned features above. To achieve a well-defined way of operations a formal definition is needed. Chapter 2.2.1 gives an overview of the formal definition of XVSM while Chapter 3.1 gives information about MozartSpaces [47], the actual reference implementation.

2.2.1 Formal Definition

The formal definition of XVSM can be found in [Cra09, eKMS08, Cra10]. It is the base for MozartSpaces. Figure 2.2 gives an overview of the architecture of XVSM. It is layer

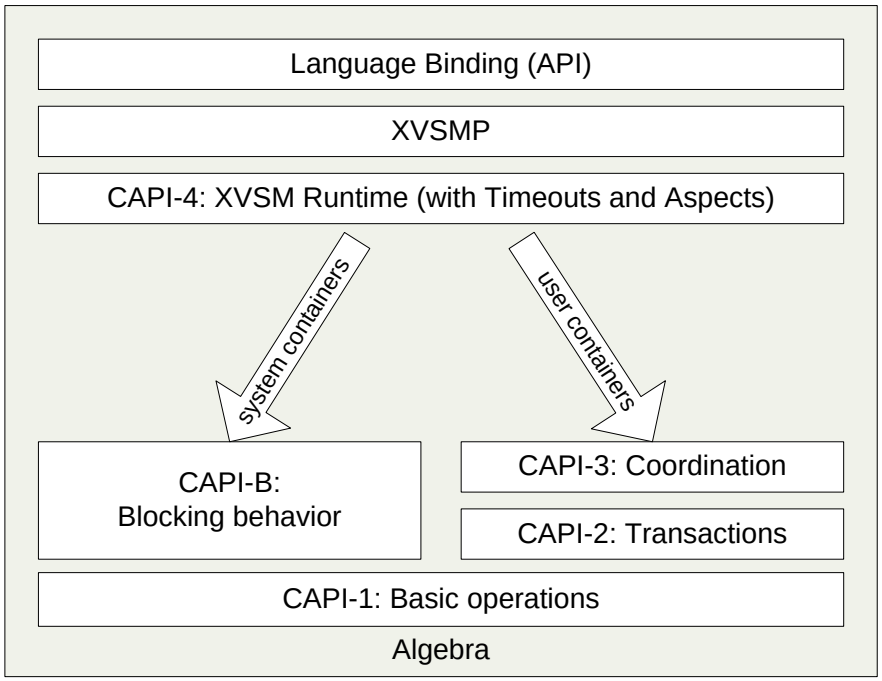

Figure 2.2: Layered architecture of XVSM [Cra10, 24]

based while each layer uses the functionality offered underneath. Each layer (Core API - CAPI) provides well-defined operations. CAPI-1 offers basic operations (read, write, take), CAPI-2 adds transactional support while CAPI-3 is responsible for coordination. All these operations are non-blocking and return immediately with different status reflecting if the operation was processed or if it was not possible (e.g. the entry is locked by another transaction). CAPI-4 (Runtime) offers timeouts and reschedules the requested operations as well as aspects. CAPI-B offers blocking behavior for basic operations and is used by the runtime. XVSMP offers interaction between different XVSM nodes based on XML.

XVSM Algebra

The XVSM Algebra defines the way elements can be accessed and manipulated. Elements can be strings or integers and collections (sequential lists and unordered bags (=multiset)). Lists offer ordered sequence of elements while bags are sets that allow duplicate elements. Collections can hold objects as well as collections. Both of them are identified by a label. Such a data structure (tree) is called xtree and is the base of the

formal definition of a space. Xtrees are defined in [Cra10] as follows:

> An xtree is either a sequence or a multiset of labeled xtrees, or an unstructured value like a string or an integer.

A list is represented by $[l_1 : e_1, l_2 : e_2, ...]$ and bags by $\langle l_1 : e_1, l_2 : e_2, ...\rangle$ while l_x are labels and e_x are elements. Anonymous elements omitting labels are allowed. For example:

$$X = [Batman : \text{“}Bale\text{“}, Opponents : \langle TwoFace : \text{“}Eckhart\text{“}, Joker : \text{“}Ledger\text{”}\rangle, Vehicle : \text{“}Batmobil\text{“}, Vehicle : \text{“}Batpod\text{“}]$$

That means that X is a multi set of four elements. The object with label “Batman” has the value “Bale” and the object with label “Opponents” is a sequence (list) that contains two elements: “TwoFace” with value “Eckhart” and “Joker” with its value “Ledger”. Two other objects with label “Vehicle” have elements “Batmobil” and “Batpod”. A subtree can be identified by paths or by an index (sequence) to access elements. Due to the fact that bags can contain elements with equal label the navigation will become in deterministic. The following example describes access to trees:

$$\begin{aligned} X.(Opponents/TwoFace) &= \text{“}Eckhart\text{“} \\ X.1 &= \text{“}Bale\text{“} \\ X.vehicle &= \text{“}Batmobil\text{“} \| \text{“}Batpod\text{“} \end{aligned}$$

The first select the sequence “Opponents” and then “TwoFace” to become the element “Eckhart”. The second uses the index notation to get the first element of the list: “Bale”. The third is in deterministic because of the property of a bag and result in either “Batmobil” or “Batpod”. With this information a space can be described as follows:

> “Basically, a space is a multiset xtree that comprises all containers located at a single site, together with their unique labels that serve as their references. A container is itself a multiset xtree containing user-generated entries with unique labels. Entries consist of labeled values called properties, which have a well-defined label and a value that can either be an unstructured value like an integer or string, or a more complex xtree.” ([Cra10, p. 11])

This principle can be used for user data as well as for meta data (e.g. transaction locks).

XVSM Query Language

To allow more precise access to the data structure than with path and indices there is a need of a query language. XVSM Query Language (XVSMQL) offers such an extended functionality. A simple query (SXQ) can be combined to a complex one by

concatenation. They are executed from left to right and the output of the previous stage is the input for the next one. There exist two different forms of SXQ: matchmakers (applied on single elements) and selectors (applied on whole input). The following selectors are defined:

- **Count operator (**$cnt(n)$**)**: Take the first n entries from a list or any n entries from a bag.
- **Sorting (**$sortup(p)$**,** $sortdown(p)$**,** $reverse()$**)**: Sort the input ascending/descending defined by path, reverse the order (sequences only) or return the input.
- **Uniqueness (**$distinct(p)$: Return unique values defined by path.

Core API (CAPI)

Each layer depends on the underlying layer and uses that functionality. CAPI 1 to 3 is non-blocking while CAPI 4 implements the runtime features that can block for a defined timeout. All CAPI operations return the result and a status:

- `OK`: Operation is succeeded.
- `DELAYABLE`: Operation is not executable at the moment. For example if a write operation tries to insert an entry in an already full bounded container.
- `LOCKED`: Operation is not executable because the data structure is locked by a transaction.
- `NOTOK`: Operation cannot be executed.

CAPI 1

The CAPI 1 layer is responsible for basic operations like read, write and take (consuming read). These operations are used for user data as well as for meta data. An update of an entry can be simulated by take followed by a write operation.

CAPI 2

The CAPI 2 layer is responsible for the transactional control. Arbitrary CAPI 1 operations can be combined to a single operation. Pessimistic locking is used as concurrency control method and “repeatable read” is used as isolation level. Transactions can be committed and roll backed by the user; sub transactions are managed by its transaction. There exist three different locks (insert/delete/read) that are responsible of how the entries and containers can be accessed by other transactions (e.g. not visible or locked)

CAPI 3

The CAPI 3 layer is responsible for the organization of the data in a container. This includes two main tasks. On the one hand defines how entries are stored and retrieved from a container. This is handled by corresponding meta data. On the other hand it is responsible for the coordination for concurrent access. There exist many pre-defined coordinators that improve usability:

- **System coordinator** (implicitly added): Check container limits.
- **Query coordinator**: Executes XVSMQL queries on a container.
- **FiFo coordinator**: FiFo ordering of the inserted entries.
- **LiFo coordinator**: LiFo ordering of the inserted entries.
- **Key coordinator**: Uses unique keys to identify inserted entries.
- **Label coordinator**: Uses not unique labels to identify inserted entries.
- **Linda coordinator**: Support Linda template matching.
- **Vector coordinator**: Store entries in an indexed list that can be used to access entries.

In addition to the predefined coordinators there is the possibility to specify custom coordinators [eKMKS09].

CAPI 4

The CAPI 4 layer includes the runtime and the aspect feature as described in [Dö11]:

> The runtime in CAPI-4 manages the timeout of operations and schedules the request processing. It also initializes the meta model where system containers are used.
>
> Aspects in XVSM are code segments that can be added dynamically during runtime and are executed when a request is processed...

The runtime handles local requests as well as remote requests. The execution result depends on the result of CAPI 3. If it is DELAYED or LOCKED it will be rescheduled and eliminated after a certain timeout if the operation does not finish. If the CAPI 3 operation result is OK or NOTOK it will be returned by the runtime immediately. The complete XVSM runtime structure is explained in [Cra10, 53].
Aspects allow the user to dynamically adapt the semantics of an operation at a specified interception point. Another operation can be executed before/after a certain operation (for example: print a log statement after a write operation).

XVSMP and Language Bindings

XVSMP (XVSM protocol) offers a programming language independent communication between different peers. Using XML for the protocol guarantees interoperability.

2.3 Mobile devices - General restrictions / properties

2.3.1 Limited Resources

In the past the limitations of mobile devices (e.g. cell phones) had a large influence on the design and scope of an application. Some years ago the available hardware had many restrictions. For instance the original iPhone [57] in 2007 had the following configuration: CPU: 412Mhz, 128MB RAM, 4/8/16GB Storage, 802.11 b/g. Comparing to the actual iPhone 5 [61] (CPU: 2x1.3GHz, 1GB RAM, 16/32/64GB Storage, 802.11 a/b/g/n), one can come to the conclusion that available space and memory as well as CPU speed should not be a considerable limitation anymore.

The limited power supply of mobile devices is another important aspect but is not in the scope of this thesis. [Zar12] gives a detailed description of the energy consumption of the MozartSpaces persistence layer on Android devices and is a good starting point for further investigations concerning energy consumption of XVSM middleware.

2.3.2 Multitasking / Background processing

Since iOS 4.0 multitasking is supported. According to the "iOS App Programming Guide" [12] an App can be in one of the five states:

- **Not running**: The application is not in memory.
- **Inactive**: Temporary state when application is in foreground but not receiving events.
- **Active**: The application is running and can receive events (e.g. multitouch gestures)
- **Background**: The application is in background and executing code. This "must not take too long" because it will be interrupted by the operating system.
- **Suspended**: The application is in background (still in memory) and not executing any code.

In figure 2.3 ([12, p. 37]) there is an overview of the possible state transitions. Most of the state changes correspond to method calls in the App delegate object. This gives the possibility to interact with the App while state transitions. Unfortunately, there

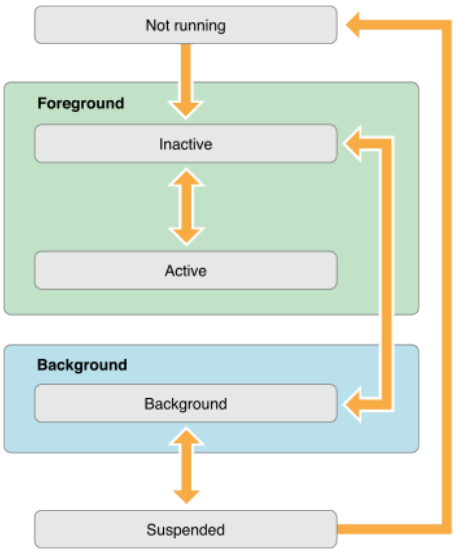

Figure 2.3: State changes in an iOS App

is no official way to handle long term running background processes except for some predefined purposes like Audio or VoIP applications.

2.4 iOS mobile devices

There are three device categories on the Apple iOS platform that are potential candidates for a XVSM implementation: The mobile phone “iPhone”, the tablet “iPad” and the portable media player “iPod touch”. These are the actual devices (End 2012) that are compatible with at least iOS 5 that is necessary for the actual implementation. Tables 2.1 and 2.2 present an overview of the different hardware. The performance will highly depend on the different hardware configurations.

Table 2.1: Overview of relevant iOS devices (iPhone)

	iPhone 3GS [1] [58]	**iPhone 4** [59]	**iPhone 4S** [60]	**iPhone 5** [61]	**iPod touch 5**^{th}gen [62]
CPU	600 MHz	800 MHz	2x800MHz	2x1.3GHz	1GHz
Capacity	8/16/32GB	8/16/32GB	16/32/64GB	16/32/64GB	32/64GB
Memory	256MB	512MB	512MB	1GB	512MB
Cellular	3G	3G	3G	3G[2]	—
Networking	802.11 b/g	802.11 b/g/n	802.11 b/g/n	802.11 a/b/g/n	802.11a/b/g/n
bluetooth	2.1+EDR	2.1+EDR	4.0	4.0	4.0

Table 2.2: Overview of relevant iOS devices (iPad / iPod touch

	iPad [52]	**iPad 2** [53]	**iPad 3**^{th}gen [54]	**iPad 4**^{th}gen [55]	**iPad mini** [56]
CPU	1GHz	2x1GHz	2x1GHz	2x1.4GHz	1GHz
Capacity	16/32/64GB	16/32/64GB	16/32/64GB	16/32/64GB	16/32/64GB
Memory	256MB	512MB	1GB	1GB	512MB
Cellular	3G optional	3G optional	3G optional[2]	3G optional[2]	3G optional[2]
Networking	802.11a/b/g/n	802.11a/b/g/n	802.11a/b/g/n	802.11a/b/g/n	802.11a/b/g/n
bluetooth	2.1+EDR	2.1+EDR	4.0	4.0	4.0

[1] used for the thesis
[2] 4G only for some mobile network operators

2.4.1 Cross platform development

Since there already exist implementations of XVSM like MozartSpaces [Bar10, Dö11, Zar12] for the Java environment and XCOSpaces [Tho08, Mar09] for the .NET environment it would be an elegant solution to reuse the already existing and well performing source code instead of reimplementing it from scratch. This would save a lot of time, increase code quality and result in a convenient multi-platform development. Since Java and all .NET languages are not officially supported for the iOS platform there is a need of a third party tool/environment to allow the execution. It has to fulfill the following requirements:

- Output must be runnable on iOS platform (iPhone and iPad).
- Reuse of the actual source code without adaptations.
- Tool/Framework must be free of charge.
- Output must be compatible to the actual MozartSpaces license (AGPLv3[3]).
- Output of source code that is compile able or binary output. They must perform concerning execution speed similar to the original binary.
- The reuse of Java code (MozartSpaces) is preferred because XCOSpaces is not fully compatible to MozartSpaces (see 3.1.4).

The following chapters present the possible solutions for the cross platform development approach.

java2objc

Java2objc [21] is a tool that converts Java programs to Objective C programs and is released under the Apache license 2.0. It converts Java source code to an equivalent Objective C source code. It is mentioned that the output is like hand written Objective C but simultaneously it should be a suggestion and may not compile. In addition the status of the project on the homepage is described as infancy. The input Java file (Appendix A.1) and Objective C output file (Appendix A.2) shows that the result is not compile able, the method calls are wrong and Automatic Reference Counting (ARC) is not supported.

[3] `http://www.gnu.org/licenses/agpl-3.0.html`

J2ObjC

J2Objc [20] is a command line tool that converts Java source to Objective C for the iOS platform and is released under the Apache License 2.0. It converts non-GUI source code like application logic. It supports exceptions, inner and anonymous classes, generic data types, threads, reflection and JUnit tests. Serialization is a heavy used feature in MozartSpaces so there is a strongly need for support. There is no Serialization support, the translation of exceptions is really bad (Objective C exceptions instead of NSError) and the project is according to the homepage "between alpha and beta quality".

In-the-box

In-the-box [19] is a project that is porting the Dalvik VM and the Gingerbread Android API (2.3.x) to iOS. This is not allowed because of the following paragraph in the Apple Developer agreement and therefore would not be accepted in the AppStore:

> 3.3.2 An Application may not download or install executable code. Interpreted code may only be used in an Application if all scripts, code and interpreters are packaged in the Application and not downloaded. The only exception to the foregoing is scripts and code downloaded and run by Apple's built-in WebKit framework.

Further the development seems to be stopped.

Monobjc

Monobjc [44] offers the tools to develop and run .NET applications under OS X and is released under the MIT/X11 license (.NET code) and GNU Lesser General Public License v3.0 (native runtime). It is actively developed but there is no support for the iOS platform.

MonoTouch

MonoTouch [65] developed by Xamarin allows .NET development for iOS devices and is released under a commercial license. The usage is not free of charge. The goal of MonoTouch is the following:

> Write your App entirely in C# and share your code on iOS, Android, Windows and Mac.

MonoTouch Apps are compiled to machine code since Apple does not allow JIT compiling.

Mono

Mono [64] developed by Xamarin is the .NET framework compatible cross-platform implementation. It allows running .NET source code (e.g. C#) on platforms among others on ARM7(s) (iOS platform) and Linux/OS X operating system. This allows running C# source code on the iOS platform. Since iOS does not allow JIT it needs AOT compiling and embedding the Mono runtime to the App. This use of Mono needs licensing by Xamarin[4] and is therefore no option:

> We only require licensing for uses of Mono and Moonlight on embedded systems, or systems where you are unable to fulfill the obligations of the GNU LGPL.
>
> For example, if you manufacture a device where the end user is not able to do an upgrade of the Mono virtual machine or the Moonlight runtime from the source code, you will need a commercial license of Mono and Moonlight.

xmlvm

XMLVM [45] is a cross-compiler tool on byte code level and is released under the GNU Lesser General Public License, version 2.1. The intention is that byte code (e.g. Java, .Net) is easier to cross-compile then source code files. The output format can be chosen between Objective C and C but former will be eliminated in the future. There exist restrictions that some classes are not implemented and therefore not useable for the cross-compiling process (e.g. `Date`, `ObjectOutputStream`, `URL`).

Conclusion

"java2objc" creates code that is not compile able and therefore not useable for the implementation. "J2ObjC" is in early stage and therefore no option for development. Hence, "In-the-box" development is stopped, it is not useable. "Monobjc" does not support iOS development and is therefore not useful. "xmlvm" does not support necessary classes and is therefore not useful. "MonoTouch" is a platform that needs to be taken into account. It has a big community and a large documentation. Unfortunately, it is not free of charge and not open source and therefore it is not used for the implementation. Similar problems occur using Mono. As described above Mono is not free of charge either. None of the mentioned possible solutions can be chosen and therefore the implementation is done by porting MozartSpaces to the iOS platform using Objective C.

[4] `http://www.mono-project.com/FAQ%3a_Licensing`

2.4.2 Jailbreak

Due to the restrictions that Apple has built into the iOS operating system (see section 8.1) some inventive people started to remove these limitations. This process is called "jailbreak". It is done by using hardware/software exploits that can be used because of security vulnerability. It allows the user to use applications that are distributed by other stores than Apple App Store. For example, the cydia App Store [27] can then be used. Due to the fact that all functions of iOS can be used after jail breaking (Apple App Store, iTunes,...) it is some kind of adding new features to the operating system. This allows for instance access to the file system, installing a command line or running processes in the background.

Legal aspects

Apple complains that jail breaking violates the Software License Agreement. In the USA it is definitely allowed since 2010 ([43]):

> "Computer programs that enable wireless telephone handsets to execute software applications, where circumvention is accomplished for the sole purpose of enabling interoperability of such applications, when they have been lawfully obtained, with computer programs on the telephone handset."

The situation in Germany is slightly different. It seems that it is not regulated [24] but Eva Dzepina from law office "Borgelt & Partner" (Düsseldorf, Germany) argued that the private use of jailbreaked devices is allowed. In Austria, there is no explicit regulation if jail breaking an Apple or other device is allowed or not. It may be in conflict with the national copyright act. This cannot be, however, subject of this thesis.

2.5 Requirements for porting MozartSpaces

To implement a MozartSpaces compatible version on a new platform there are requirements that need to be fulfilled by the operating system. Table 2.3 gives a summary of the necessary features/technologies of the operating systems. Since iOS is a UNIX based operating system it supports sockets, threads/locks similar to Android OS. The SQLite database is included and used extensively in iOS. The background processing is restricted (see 2.3.2) but there is a workaround and iOS 7 allows it anyway. There exists an object oriented programming language (Objective C) as counterpart to Java on iOS. The communication can be established with WLAN interface.

[5] Not included in iOS but can be built like on any other UNIX platform

[6] Not allowed but workaround exists. iOS 7 supports background processing.

Table 2.3: Requirements for porting MozartSpaces on the iOS platform

Requirement	Technology	Android	iOS
Communication	Socket	x	x
Concurrency	Threads/Locks	x	x
Persistency	SQLite	x	x
	Berkeley DB	x	x[5]
Background processing	Daemon	x	x[6]
Programming Framework	Object oriented Programming Language	Java	Objective C
Configuration	Programmatically	x	x
	XML	x	x
Communication interface	WLAN	x	x

2.6 Communication in heterogeneous systems

2.6.1 Serialization

In [HJR+03] is serialization and deserialization defined as follows:

> "Object serialization is the process of writing the state of an object to a stream. Deserialization is the process of rebuilding the stream back into an object."

Data exchange in heterogeneous systems needs special treatment because standard exchange formats of MozartSpaces like Java serialization do not work in iOS environment. Therefore there is a need of a format that can be handled in Java as well as in Objective C. It [SM12] is a comparison of different serialization formats: XML [51], JSON[7], Thrift[8], and ProtoBuf[9]. The first two are text-based while the latter two use a binary format. Advantages of XML and JSON are that they are widely used, human readable and there exists libraries for many programming languages. The largest disadvantage for mobile devices is the big markup overhead. Thrift and ProtoBuf are extremely lightweight, fast to serialize and deserialize and are therefore well useable for mobile devices. The two large disadvantages of the binary formats are that they are not human readable and therefore hard to debug and that they are specific for programming languages. The paper compares the serialization formats concerning serialization speed, data size, and usability. Two different test objects were used for the evaluation:

[7] `www.json.org`
[8] `thrift.apache.org`
[9] `code.google.com/p/protobuf`

firstly a heavy text object ("Book" object in the table) and secondly a heavy number object ("Video"). Table 2.4 shows the different serialization size. The binary formats are the smallest and XML is by far the biggest. Table 2.5 and 2.6 depict the serialization and the deserialization performance. The binary formats are the fastest while XML is the slowest. The last criterion is the usability. This includes human readability, platform compatibility and available documentation. XML and JSON cover all criteria very well while Thrift and ProtoBuf are not human readable and only supported on a limited number of programming languages. ProtoBuf does not support Objective C natively but there is an external implementation called protobuf-objc [10] which is well documented. Thrift has the drawback of missing documentation but supports Objective C by default. Another interesting criterion would be CPU usage. Since it is closely related to the performance it is not explicitly mentioned.

Table 2.4: Serialized size in bytes [HJR+03]

	XML	JSON	ProtoBuf	Thrift
Book	873	781	687	720
Video	231	139	59	92

Table 2.5: Average Serialization Time in ms [HJR+03]

	XML	JSON	ProtoBuf	Thrift
Book	22.842	4.177	2.339	2.315
Video	17.884	4.097	1.800	1.747

Table 2.6: Average Deserialization Time in ms [HJR+03]

	XML	JSON	ProtoBuf	Thrift
Book	7.908	1.199	0.298	0.732
Video	6.742	0.755	0.197	0.310

For a mobile middleware the performance and the data size are the most important properties. The importance of serialization and deserialization performance is obvious but also the size is important. Since using wireless/3G communication bandwidth is a limited resource. The dependency on external libraries including the licenses must also be taken in concern.

[10] `github.com/booyah/protobuf-objc`

CHAPTER 3

Related Work

There are many different middleware implementations based on the Tuple Space communication model by David Gelernter ([Gel85]) with different features and points of view. Some are related to business applications (e.g. GigaSpaces [34]) while others run on embedded devices like wireless sensor networks (e.g. Agilla [FRL09]). This shows the potential of this technology in many different fields of application. This thesis is about implementations that have at least the following features:

- Compatible to XVSM reference implementation, MozartSpaces 2.x concerning:
  - A) Operations
  - B) Transaction
  - C) Coordination
  - D) Aspects
  - E) Communication protocol
  - F) API semantics
  - G) Persistency operations
- Runnable on iOS platform

The following actual implementations are potential candidates to fulfill the above requirements.

3.1 Actual Implementations

3.1.1 Java implementation - MozartSpaces

The actual MozartSpaces implementation [47] is based on two master theses, which are based on the formal definition, described in 2.2.1 and is the reference implementation of the XVSM technology. The design and implementation of Operation, Transaction and Coordination is described in [Bar10] and the design and implementation of Runtime, Protocol and API is explained in [Dö11]. There were some changes like combining CAPI 1 to 3 in a single layer (CAPI 3) due to performance considerations or adding "read committed" and "repeatable read" isolation levels to the implementation. Additionally, a persistence framework was added [Zar12]. It supports storing entries based on Berkeley DB [35] and SQLite [50] and allows continuing work after shutdown and restart of MozartSpaces. MozartSpaces is the Java implementation of XVSM with many useful features like aspects, different coordinators (e.g. FiFo, Random, Linda and Query) or authorization features using an access control model [CDJ+13]. A replication mechanism ([Hir12]) to avoid loss of data and distributed transactions ([Brü13]) to increase the level of data consistency is also supported. Due to the feature richness, a wide field of application exists. For example, the SILCA framework [eKMS+12] used for load clustering is based on MozartSpaces.

3.1.2 Java implementation - MozartSpaces running on Android

Since Android OS Apps are based on Java the actual MozartSpaces implementation can be easily adapted to be compile able for Android OS based smartphones. The design remains the same but there is a need for extensions like adapting for different API levels or improving the communication implementation for cellular network. The work is still in progress and described in [Floon]. The features of MozartSpaces of the desktop version can also be used for the Android OS version.

3.1.3 .NET implementation - TinySpaces

TinySpaces [Mar10] is an XVSM implementation based on the .NET Micro Framework. It can be used on embedded system, e.g. sensor networks. An advantage is the wide applicability of this implementation because of the hardware independency of the runtime environment. It can be used on many embedded systems using different architectures. Since the code must be interpreted at runtime there might be problems with performance and energy consumption. Although TinySpaces is based on the formal definition of XVSM [Cra10] it is not fully compatible to MozartSpaces. Aspects are implemented in a different way as described in [Mar10]:

> "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 ... is used as a source here."

The communication functionality is completely separated from the runtime and is configurable. A corresponding XML serializer to the new Java/Objective C XML serializer would allow communication.

3.1.4 .NET implementation - XCOSpaces

XCOSpaces [Tho08, Mar09] is a .NET implementation that interoperates via XML with former version of MozartSpaces (1.x) but it is not fully compatible. The following .NET types are supported: `bool`, `byte`, `int`, `long`, `float`, `double`, `string`, `DateTime`, `byte[]` and `Uri`. Objects cannot be serialized and are therefore not used for communication but there is the possibility to use a tuple that consists of a combination of data entries. The design is not based on the formal definition in [Cra10] and is therefore not compatible to MozartSpaces 2.x. For example, different runtime model and different layering with other API semantics.

XCOSpaces allows implementation of a XML serializer similar to the Java/Objective C version and could then allow communication to MozartSpaces or the new Objective C implementation.

3.1.5 iOS implementations

At the moment there exists no publication or implementation on the internet that deals with the iOS platform in combination with the requirements mentioned above. Therefore the implementation proposed in this thesis course tries to fill this gap.

3.1.6 Summary

There exists no XVSM implementation that is fully compatible to MozartSpaces as shown in table 3.1 and running on the iOS platform. The different criteria mentioned above are compared on which level the implementations can interact. As it has been shown none of the existing implementations that fulfill the requirements are runnable on the iOS platform.

	Programming language	Compatibility							iOS platform
		A	B	C	D	E	F	G	
MozartSpaces	Java	Yes	Yes	Yes	Yes	Yes[17]	Yes	Yes	No
MozartSpaces (Android)	Java	Yes	Yes	Yes	Yes	Yes[17]	Yes	Yes	No
TinySpaces	.NET	Yes[11]	Yes[11]	Yes[11,12]	No	No[18]	No	No	No
XCOSpaces	.NET	No	No	No	No	No[13,18]	No	No	No
new Objective C implementation[16]	Objective C	Yes	Yes	Yes[12]	Yes	Yes[14,17]	Yes	Yes[15]	Yes

Table 3.1: Overview of different XVSM implementations

[11] Combined in MozartSpaces (CAPI-3) but layered in TinySpaces
[12] Subset of MozartSpaces coordinators
[13] Actual implementation compatible to MozartSpaces 1.x with restricted number of datatypes.
[14] Aspects cannot be serialized
[15] InMemory persistence backend implemented
[16] Implemented in this thesis
[17] Using new XML serializer - available in Java and Objective C
[18] A corresponding .NET (Micro)Framework implementation of the new XML serializer would allow communication

CHAPTER 4

Use Cases

An XVSM implementation that is fully compatible with MozartSpaces brings new opportunities for different kind of scenarios. It offers simplifications for the developers to create peer-to-peer scenarios with little lines of code in the homogenous iOS devices as well as in the heterogeneous communication with MozartSpaces. The following chapters summarize the most relevant ones.

4.1 Intra-App communication

This first possible area of application is using XVSM for communication inside a single App between different threads. This brings features like persistency and a substitute of concurrent data structures to an iOS application including the coordination between threads. In addition data organization in the coordinators (e.g. Random, FiFo) can save lots of code lines. Since the iOS SDK does not support concurrent data structure such as Java with the Concurrent package (e.g. `ConcurrentHashMap`) this simplifies development of multithreading Apps. The persistency feature supplements the standard persistency possibilities (SQLite, CoreData). Figure 4.1 outlines communication between two threads using XVSM.

4.2 Inter-App communication

Including using cases presented in section 4.1 there is the possibility of inter-process (App) communication. Since XVSM is running in the background this will offer the full coordination abilities as known as on desktop computers. Figure 4.2 outlines the communication between two different processes (Apps) using XVSM. This allows inter process communication on a very high level.

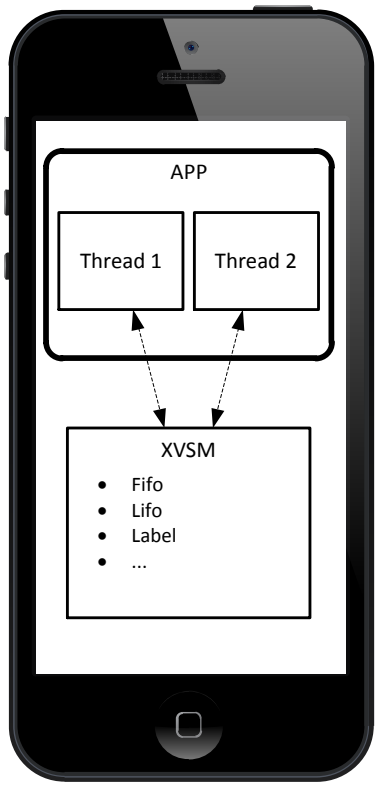

Figure 4.1: Intra-App communication using XVSM

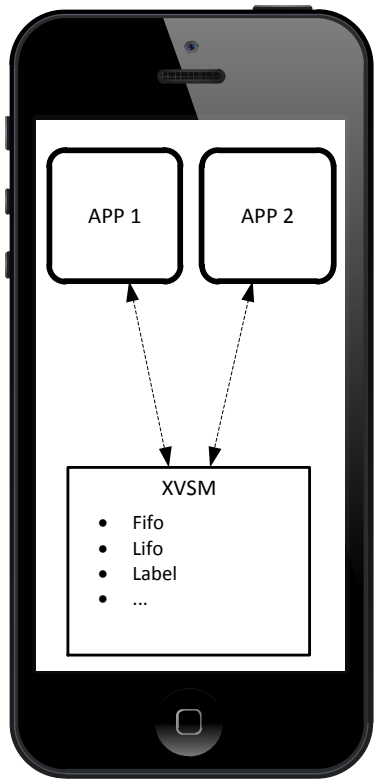

Figure 4.2: Inter-App communication using XVSM

4.3 Remote communication

WLAN can be used for communication on mobile devices because it is working fast, it is easy to configure and it allows communication with many different computer systems. This can be used for every application (e.g. games or business application) as long as the devices are in range of a WLAN access point. Then the XVSM design (e.g. loose coupling) offers its advantages. (Mobile) devices sharing a combined view on data can be realized with only few lines of code.

Cellular communication (3G/4G) cannot be used at the moment because of the usage of NAT (Network Address Translation) of the network operators. A well-known technique (STUN (Session Traversal Utilities for NAT)) was tested (5.3.2) but it was not working.

CHAPTER 5

Implementation

Since using the existing Java source code cannot be used for the iOS platform and the reuse of the source using different technologies as described in chapter 2.4.1 is not possible there comes a need of a new implementation or of porting source code of MozartSpaces (Java) to the iOS platform (Objective C). The existing code is well performing and tested by many developers and therefore it is a logical choice starting a porting process to become a compatible XVSM implementation to the iOS environment.

The following chapter describes important aspects concerning the porting process (see 5.1), issues developing on the iOS platform (see 5.2) and implementation details (see 5.3) including a How-To using the new Objective C implementation.

5.1 Porting process

5.1.1 General aspects

Porting Java source code to Objective C leads to some changes in the code base because of differences in the programming language (e.g. different features of collections concerning multithreading) and the environment (e.g. garbage collection in Java versus automatic reference counting in Objective C) but the interface definition is very similar. Especially the most upper interfaces like `Capi` are fully compatible with the Java implementation and will look familiar to MozartSpaces users. Most parts of the implementation are compatible as well. This brings some advantages for the developers and designers: First of all the well-tested and performing source code will be reused. This increases the implementation quality because many problems are already solved in the Java implementation. Secondly, future adaptations needs to be designed once and can then be used for both platforms. This applies for bug fixes as well as for new features.

Thirdly, the integration tests of MozartSpaces can be used to check compatibility by executing against the Objective C implementation. This allows future development as easy as possible because the Objective C code base has similar code quality as MozartSpaces. Fourthly, the formal definition of MozartSpaces is reused and therefore the semantic is clearly defined.

5.1.2 Restrictions

Due to the fact that MozartSpaces became a large and feature rich application some compromises needed to be done for the proof of concept prototype of the new implementation performed in this thesis. The following list summarizes the main differences between MozartSpaces and the new implementation for the iOS environment:

- No authorization functionality at all;
- The persistence layer is prepared (and partly implemented) but not tested and working (including any caching mechanism). Actually there works the InMemory configuration. For iOS exists well-documented official support for SQLite (already integrated in iOS) but only little information using Berkeley DB. Therefore SQLite will be a good starting point for persistence.
- Limited number of coordinators compared to MozartSpaces: AnyCoordinator, LabelCoordinator, KeyCoordinator, FifoCoordinator, LifoCoordinator, RandomCoordinator
- "HelloSpace" and the application scenario are the only example applications.
- Documentation of source as well as tutorials need to be taken from MozartSpaces and be used accordingly.
- Standalone space using some kind of background processing (see 5.2.1)
- Some features that are implemented in different ways and can be selected in MozartSpaces are only implemented in one way. For example there exists no "NonPollingTimeoutProcessor". The polling equivalent `XMPollingTimeoutProcessor` is used.
- Configuration is tested only programmatically and not using a configuration file.
- Aspects cannot be serialized at the moment.
- No notification mechanism implemented right now.
- No ReST implementation.

- No equivalents for JAXB, kryo or XStream implementations.
- The serialization mechanism used to communicate between MozartSpaces and the new Objective C implementation has performance drawbacks.
- The meta model functionality is not implemented.

5.1.3 Porting details

The project is developed on a MacBook pro (Mid 2009) with 2,53GHz and 8GB RAM. As operating system is MacOS X Lion (10.7) used with the latest updates (July 2013). The mobile device is an iPhone 3GS with iOS 5.1.1. The development environment is Xcode 4.6.3 with the Apple LLVM 4.2 compiler. SDK 5.1 and 6.1 is used including the according iPhone/iPad simulators.

Apple only supports developing using the Xcode environment. This includes using a Mac. Linux or Windows is not supported officially. There exist some solutions using OS X in a virtual machine[19] and then using Xcode. Since Apple does not allow running OS X on non-Apple machines[20] it is prohibited. Other solutions like modifying the SDK or jail-breaking are maybe illegal (see 2.4.2) or bring at least the risk that the App is rejected by Apple when bringing it to the AppStore. Alternative IDEs like AppCode[21] running on a Mac and support other features uses the Xcode environment and do not get in conflict with Apples licenses.

Coding convention

In principle, the generated source code for this thesis fit to the Apple Coding Guidelines for Cocoa [7]. This is necessary because many of the properties of Objective C depend on conventions[22] and so it needs to be defined clearly. There exists no namespaces in Objective C and therefore it is common to use prefix for every project to avoid naming conflicts. The "XM" prefix is used. A MozartSpaces class called `FifoCoordinator` becomes `XMFifoCoordinator` in the Objective C implementation. The typographic convention is camel-casing (e.g. capitalizeFirstLetterOfEachWord). Because of the different naming styles of Java and Objective C, the method naming is little different. Listing 1 presents the difference in the different naming styles. When translating Java code to Objective C the guideline in [Buc10] is used. For instance, an `ArrayList` become an `NSArray` if it is read only otherwise `NSMutableArray`. If the Java feature is not supported it will be simulated to have a close relation (e.g. inner classes

[19] `http://www.macbreaker.com/2012/02/lion-virtualbox.html`

[20] `http://www.apple.com/legal/sla/`

[21] `http://www.jetbrains.com/objc/`

[22] All methods including alloc, copy, mutableCopy or new delegate the release responsibility to the calling method

of Java will be translated to standard Objective C classes). The following features are available for Java but not for Objective C:

- Method overloading: Different method signatures are used to deal with that problem.
- No generics: A type of `id` is used and all collections are heterogeneous.
- No annotations: Annotations are skipped and handled implicitly.

```
public ContainerReference lookupContainer(final String name, final
      URI space, final long timeoutInMilliseconds, final
      TransactionReference transaction)
```

```
- (XMContainerReference *)
      lookupContainerWithName:(NSString *)name
                        Space:(NSURL *)space
        TimeOutInMilliseconds:(long long int)timeoutInMilliseconds
         TransactionReference:(XMTransactionReference *)transaction
                        Error:(XMError **)error
```

Listing 1: Difference between Java and Objective C syntax

Another aspect is the object creation. Each class has a convenience constructor (e.g.`+(XMCapi *)capiWithCore:(id <XMXaviCore>)core`) to create an object with one meaningful class method call.

Testing

GHUnit [28] is used as test framework. It supports command line execution very well and also outputs error messages clearly. Each test class needs to be inherited from `GHTestCase`. It support features like setup and teardown for each test case as well as for a test class as known from JUnit [40]. Each test method needs to have a "test" prefix. All test classes have the "Test" prefix and the interface block and implementation block are written in one file.

Concurrency

All immutable collections like `NSArray`, `NSDictionary`,... are thread safe but their mutable equivalent (e.g. `NSMutableArray`, `NSMutableDictionary`,...) are not. Therefore, a manual locking mechanism must be used. Objective C supports many different locking mechanism as described in [16]. The following types are used depending on the situation:

- POSIX Mutex Lock: Basic mutex - e.g. used in `XMNativeLock`

- SpinLock: Used when locked sequence can be executed in a short amount of time - e.g. used in `XMDefaultSubTransaction`
- `@synchronized` block: Slowest but most convenience mechanism. Releases the lock when leaving `@synchronized` block (e.g. after throwing exception or leaving the method) - e.g. used in `XMDefaultTransaction`

Error handling / Exceptions

Exception handling is slow in Objective C and reserved for exceptional conditions and not for the standard program flow like when a container name is not available any more. For this situations an `NSError` object will be used and the methods have an indirect reference to this error object. All exceptions of MozartSpaces inherited of `Exception` become an error object `XMError` a subclass of `NSError`. All runtime exceptions inherited of `RuntimeException` become an Objective C exception. It is used for unexpected situations like array index out of bounds error. This decision was made because all MozartSpaces exceptions are caught after a method call and therefore easy to translate to an error object. The MozartSpaces runtime exceptions are exceptional conditions and therefore translated to Objective C exceptions. This concept has the advantage to get the speedup of using error objects for recoverable problems and Objective C exceptions for unexpected program behavior.

Memory Management

The memory management feature "Automatic Reference Counting - ARC" is used for all classes. This is a recommendation by Apple since invention of Xcode 4.2, Apple LLVM 3.0+ compiler. In principle, in ARC the reference counting is the same as for manual reference counting except that the retain and release statements are inserted at compile time automatically by the compiler [15]. The complete technical specification of ARC is shown in [23]. Apple promises speed up compared to manual reference counting and also compared to garbage collection that is not support by the iOS environment.

5.2 iOS issues

5.2.1 Background processing

As described in 2.3.2, there is no official way to run long-term background processes/daemons. Nevertheless, four different (undocumented) approaches were tried out to ascertain that background processing is no option. Based on the keynote of WWDC 2013 Apple will change its focus and allow background processes in iOS 7.

Background processing using Finite-Length task

Before an App is suspended it will get into background state. Then the `applicationDidEnterBackground:` method gives the opportunity to run finite length tasks. In listing 2, there is an example how to use this feature. At the beginning a background task must be registered (`beginBackgroundTaskWithExpirationHandler:`). After that a thread can be used to run the background task. At the end the application must be informed that the background has finished otherwise the complete App will be killed by the operating system. In test runs there were 600 sec to finish the background task. Another possibility would be to nest many finite length tasks. Unfortunately this is does not work either. There seems to be one counter per App.

```
- (void)applicationDidEnterBackground:(UIApplication *)application {
  __block UIBackgroundTaskIdentifier bgTask = [application
         beginBackgroundTaskWithExpirationHandler:^{
       [application endBackgroundTask:bgTask];
       bgTask = UIBackgroundTaskInvalid;
  }];

  dispatch_async(dispatch_get_global_queue(
         DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
       ^{
           long long int i = 0;
           //finite task
           while (i < 100) {
               NSLog(@"%lld - BackgroundTimeRemaining: %.0f sec", i++,
                       [UIApplication sharedApplication].
                     backgroundTimeRemaining);
               sleep(1);
           }
           [application endBackgroundTask:bgTask];
           bgTask = UIBackgroundTaskInvalid;
       });
}
```

Listing 2: Background processing using finite length task

Background processing using “audio” environment

Another promising approach is defining a specified profile to the App. Among other there exists the profile “audio”. It allows playing audio in the background. It must be defined in the App `Info.plist`:

```
<key>UIBackgroundModes</key>
<array>
  <string>audio</string>
```

```
</array>
```

Listing 3: Background processing using “audio” environment

This should allow defining a process to run in background. But this does not work when creating a thread. The App will be suspended after some time.

Background processing using fork

Another possibility is using `fork()` to create a new process as described in listing 4. Unfortunately this call is not allowed in iOS.

```
pid_t pid;
switch (pid = fork()) {
  case -1:
    NSLog(@"could not create fork - CHILD!");
    return;
  case 0:
    NSLog(@"child process");
    break;
  default:
    NSLog(@"parent process");
    break;
  }
return;
```

Listing 4: Background processing using `fork()`

Background processing using dummy AVAudioPlayer

The only working solution is the usage of a dummy audio that uses the “audio” environment in combination with a player that play an empty file with zero volume. The first step is the configuration like in section 5.2.1 followed by starting a dummy player with the included thread as shown in listing 5.

```
- (void)viewDidLoad {
  [super viewDidLoad];
  NSURL *audioFileLocationURL = [[NSBundle mainBundle] URLForResource
        :@"dummy" withExtension:@"mp4"];

  NSError *error;
  audioPlayer = [[AVAudioPlayer alloc] initWithContentsOfURL:
        audioFileLocationURL error:&error];
  [audioPlayer setNumberOfLoops:-1];

  if (error) {
    //error handling
  } else {
```

```
    [[AVAudioSession sharedInstance] setCategory:
          AVAudioSessionCategoryPlayback error:nil];
    [[AVAudioSession sharedInstance] setActive:YES error:nil];
    [audioPlayer prepareToPlay];
  }
}

- (IBAction)startPressed:(id)sender {
  dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
    ^{
      long long int i = 0;
      while (YES) {
        NSLog(@"%lld - BackgroundTimeRemaining: %.0f sec", i++, [
              UIApplication sharedApplication].
              backgroundTimeRemaining);
        sleep(1);
      }
    });
  [audioPlayer setVolume:0];
  [audioPlayer play];
}
```

Listing 5: Background processing using dummy AVAudioPlayer

A problem to distribute the App through the AppStore is the fact that using multitasking services not for the intended meaning (audio, location, voip, newsstand-content, external-accessory, bluetooth-central) is not allowed by the Developer agreement:

> 3.3.6 You may use the Multitasking services only for their intended purposes as described in the Documentation.

5.3 Implementation details

5.3.1 Serialization

At the moment there exists two serializers for the new Objective C environment. The native Built-in serializer (`XMBuiltinSerializer`) and the XML serializer (`XMMzsXaviSerializer`) for heterogeneous environments. Both implement the protocol `XMSerializer` and can therefore be used for communication process as well as for persistence operations.

Built-in serialization

The object serialized by the Built-in serializer must implement the `NSCoding` protocol. It defines two methods: one for encoding (`-(void)encodeWithCoder:(NSCoder`

`*)aCoder`) and one for decoding
(`-(id)initWithCoder:(NSCoder *)aDecoder`). The `NSKeyedArchiver` take care of the encoding process and `NSKeyedUnarchiver` is responsible for decoding. This is the standard way of serialization recommended by Apple [6].

XML Serialization

Although both binary formats described in [SM12] fit well to the requirements (first best speed then best data size) XML was chosen for this implementation to support multi platform communication. It is on the one hand human readable, easy to understand and there exist many parsers for Java and Objective C that can be used. The principle of designing the serializer in Java and porting to Objective C was done like for MozartSpaces. Especially the interfaces are identical. This allows that optimization on one environment and leads to optimization in the other environment. As the XStream[23] output the serializer does not comply with any XML Schema Definition. Every class that implements the `MzsXaviSerializerMarker` respectively `XMMzsXaviSerializerMarker` and can be serialized. Because of the differences of the programming languages and for simplification some other restrictions were made:

- Neither the keywords of Java nor the keywords of Objective C are allowed for names.
- In collections there is only one object type allowed.
- No pointers are allowed in Objective C.
- No modifier like unsigned are allowed.
- Only a restricted number of data types that can be easily mapped to XML are supported as listed in table 5.1.
- All collections in the messages (responses and requests) are handled as non-concurrent lists, sets and maps.
- Aspects can be serialized.
- Equal classes in both environments must exist and have the corresponding variable definition (including data types).
- The class naming convention is as following: Java class name with an “XM” prefix for Objective C. E.g. Java class `NiceClass` is named `XMNiceClass` in Objective C.

[23] `http://xstream.codehaus.org`

The following features are implemented in the actual serializer:

- Resolve class inheritance.
- Serialize all supported attributes including private and protected attributes.
- No mapping between objects needs to be explicitly added.
- Output in valid XML ([51]) format not conforming to any XML schema definition.
- Support of circular references in an object graph.
- Support of multiple references to the same object.
- Mapping of Java exceptions to corresponding Objective C errors respectively exceptions and vice versa.

Table 5.1: Mapping Java to Objective C datatypes

Java type	Objective C type
`boolean`	`BOOL`
`int`	`int`
`long`	`long long int`
`double`	`double`
`String`	`NSString`
`StringBuffer`	`NSMutableString`
`ArrayList`	`NSMutableArray`
`LinkedList`	`NSMutableArray`
`Vector`	`NSMutableArray`
`HashSet`	`NSMutableSet`
`HashMap`	`NSMutableDictionary`
`URI`	`NSURL`
`Collections$EmptyList`	`NSMutableArray`
`MzsXaviSerializerMarker`	`XMMzsXaviSerializerMarker`

5.3.2 Cellular communication

A possible solution to solve the NAT problem is using a STUN implementation. For Objective C there exists STUN-iOS [41]. Running it for checking Austrian mobile

network carriers comes to the result presented in table 5.2. In Austria there are three cellular network operators: A1 Austria, T-Mobile Austria and Hutchison Drei Austria with their different labels. It was only possible to check A1 Austria ("Bob") and Hutchison Drei Austria ("Orange"). Both use symmetric NAT and can therefore not be used with STUN. Since the allowing cellular communication is not a main topic of this thesis there is other research necessary.

Table 5.2: NAT behavior of Austrian network operators

SIM Carrier	Mobile Country Code	Mobile Network Code	Symmetric NAT
Orange	232	05	yes
Bob	232	11	yes

5.3.3 Implementation details

Directory structure

The directory structure of the new Objective C implementation depends on Xcode structure of defined targets. Each target gets per default its own directory with its entry point (main function including `AppDelegate` for iOS environment). To allow compiling for iOS and for OS X a separate directory for the classes is used. Figure 5.1 presents the actual file structure. `bin`, `build`, `doc`, `etc`, `Frameworks` and `usr` are explained in the figure. The application scenario is splitted into 2 directories: `MzsXaviChatiOS` has the implementation classes for the iOS environment and `MzsXaviChatMac` has the implementation classes for OS X. All benchmark classes are stored in `Performance`, the entry points for ARC is in `PerformanceMacARC` and for GC in `PerformanceMacGC`. `Testing` has all test classes, `TestingiOS`, the entry points for iOS are in `TestingiOS` and `TestingMac` for OS X.

Project structure

Xcode offers a logical grouping feature that is independent of the directory structure in the file system. It offers grouping feature of files/classes to keep the overview. The structure is closely related to MozartSpaces' Java package structure. Additionally there are some groups for external source code (e.g. CocoaLumberjack) and a group for the new serializer (mzsxaviserializer). Figure 5.2 gives an overview of the actual structure.

[24] `http://www.doxygen.org`
[25] `http://cloc.sourceforge.net`

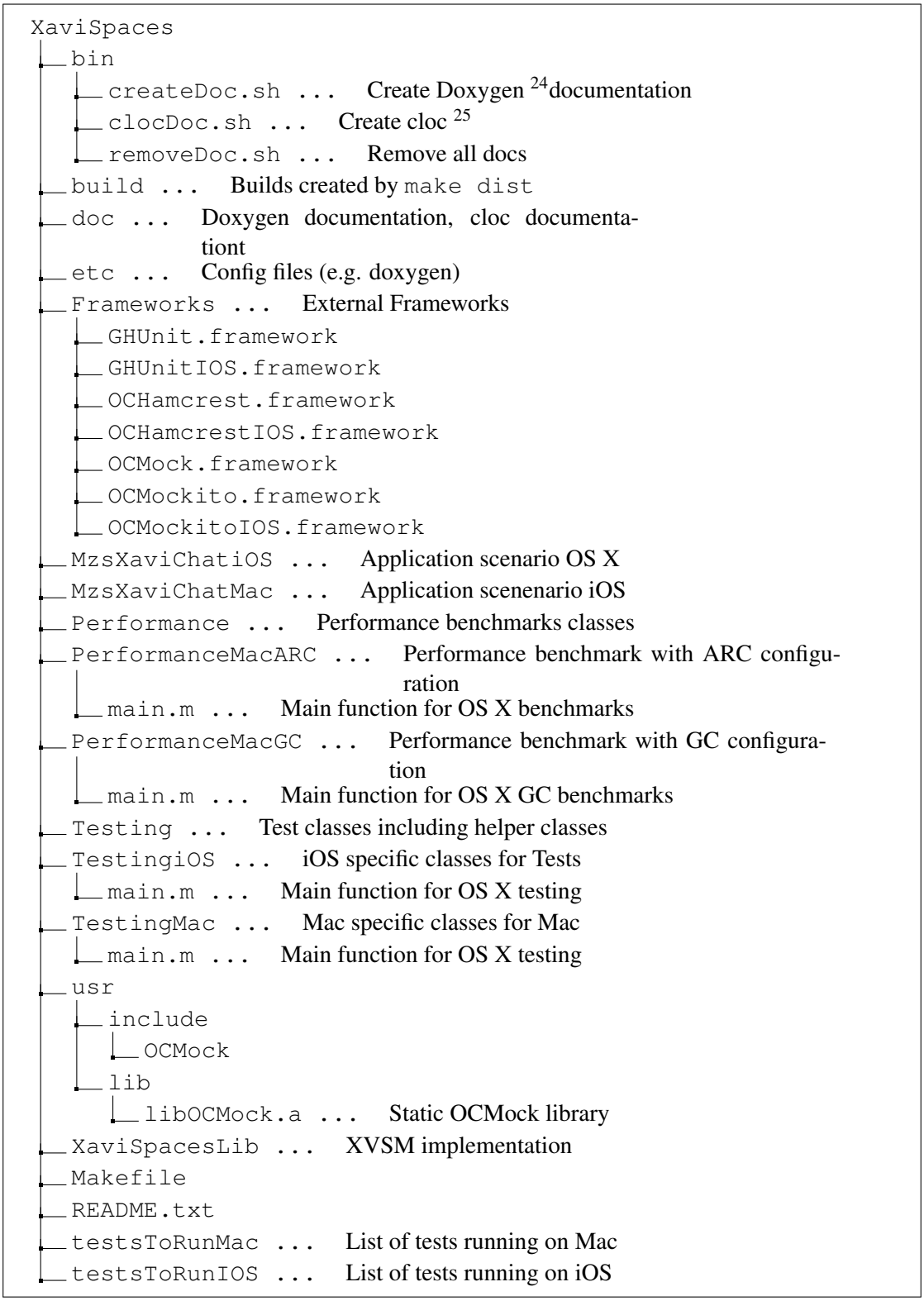

Figure 5.1: Directory structure of the new Objective C implementation

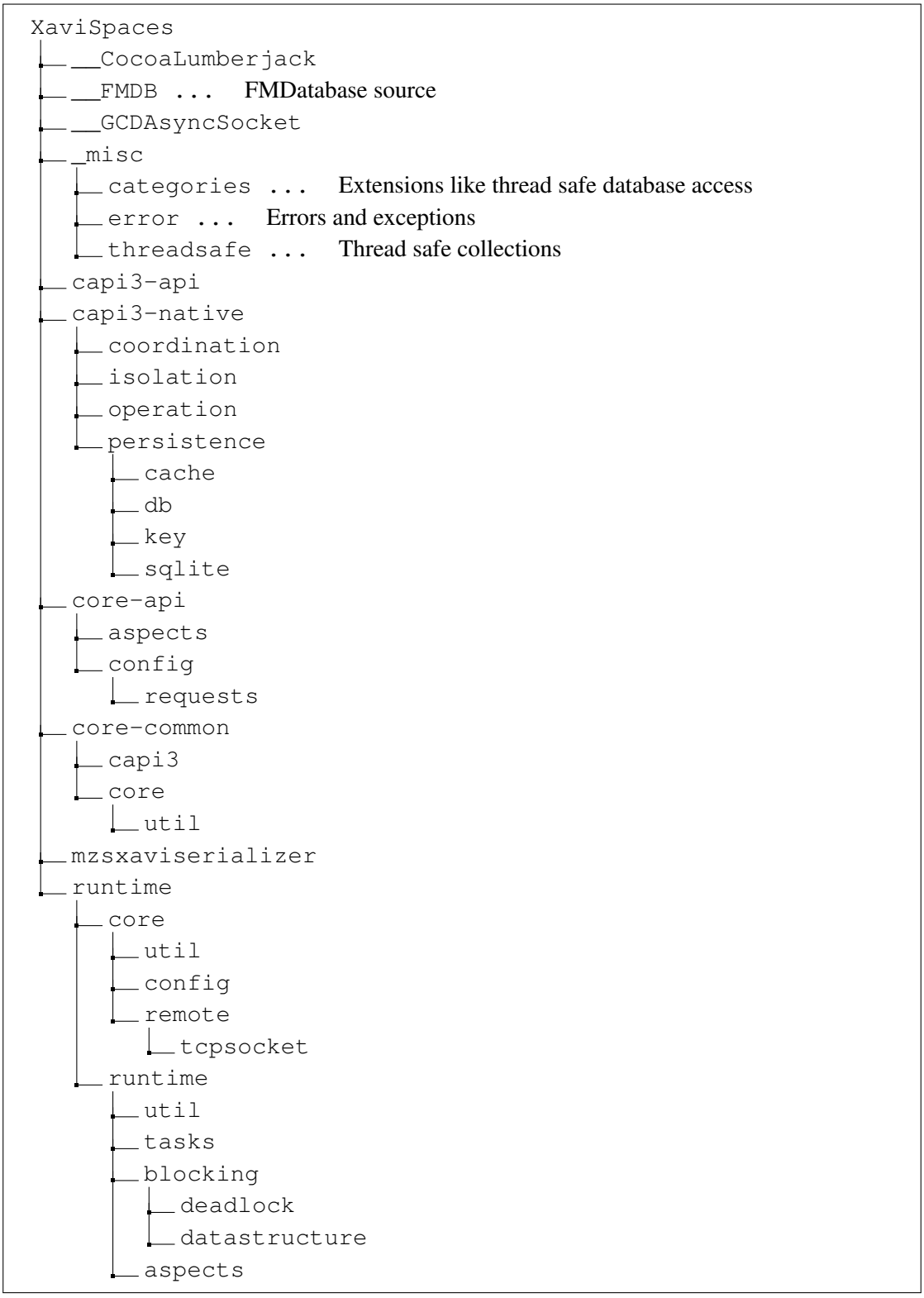

Figure 5.2: Logical structure of the new Objective C implementation

Makefile

Apart from Xcode building features a Makefile is used to get a command line interface for the build and execution process. It uses the shell variables described in 5.3.3 to get fine graded access to different build and running configurations. Appendix A.3 gives the content of the Makefile with description as comments.

Dependencies

Despite of the focus on little dependencies the new implementation needs some frameworks and libraries. Table 5.3 show the dependencies running the tests for iOS as well as for OS X. Table 5.4 show the dependencies when using the static library of the new Objective C implementation[26].

Table 5.3: Testing environment dependencies

Name	Type	iOS	OS X	Reference
`UIKit`	framework	x	-	SDK
`CFNetwork`	framework	x	-	SDK
`libsqlite3`	dynamic library	x	x	SDK
`Security`	framework	x	x	SDK
`CoreGraphics`	framework	x	-	SDK
`OCMockitoIOS`	framework	x	-	[32]
`OCHamcrestIOS`	framework	x	-	[31]
`libOCMock`	static library	x	-	[42]
`GHUnitIOS`	Framework	x	-	[28]
`Cocoa`	Framework	-	x	SDK
`OCMock`	Framework	-	x	[42]
`GHUnit`	Framework	-	x	[28]
`OCMockito`	framework	-	x	[32]
`OCHamcrest`	framework	-	x	[31]
CocoaLumberjack	source files	x	x	[46]
FMDatabase	source files	x	x	[29]
GCDAsyncSocket	source files	x	x	[30]

[26] CocoaLumberjack, FMDatabase and GCDAsyncSocket are not available as framework or library and are therefore compiled into the static library.

Table 5.4: New Objective C implementation dependencies

Name	Type	iOS	OS X	Reference
libXaviSpacesLibiOS	static library	x	-	
libXaviSpacesLibMac	static library	-	x	
Foundation	framework	x	x	SDK
UIKit	framework	x	-	SDK
CFNetwork	framework	x	x	SDK
libsqlite3.dylib	dynamic library	x	x	SDK
Security	framework	x	x	SDK
CoreGraphics	framework	x	-	SDK
AppKit	framework	-	x	SDK

Testing

All unit tests as well as integration test can be executed from Xcode using different targets and schemes or on the command line using the *Makefile*. The `testsToRunMac` and `testsToRunIOS` files consist of lists of test classes for the corresponding development environment. The following shell environment variables are available (only OS X):

- `GHUNIT_SUITE`: Defines either using a test suite or running all tests.
- `GHUNIT_CLI`: Defines if tests run on the command line or using a GUI.
- `GHUNIT_TEST_GUI`: Defines if tests has output on GUI or on the command line.
- `GHUNIT_SUITE_TEST`: List of tests in a test suite.
- `GHUNIT_SUITE_FILE_TESTS`: File name with list of tests (e.g. `testsToRunMac`) (# introduces comments in the list)
- `GHUNIT_INTEGRATION`: Defines if tests environment is used for integration tests with MozartSpaces.
- `GHUNIT_INTEGRATION_TYPE`: either "`OBJC_JAVA`" or "`JAVA_OBJC`". Defines the direction of the integration test. The first is the creating and the second the reading environment.
- `GHUNIT_INTEGRATION_OBJECT`: Defines the message type that is sent (e.g. for a "RollbackTransactionRequest": `MsgRollbackTransactionRequest`)

- `GHUNIT_STANDALONE`: Using the test environment to start a standalone space, which can be used by an external core.

The classes consist of unit tests except the `TestIAllIntegrationTest` class. It runs all integration tests with their necessary setup.

5.3.4 Interface description for users

The following section gives an overview of the programming interfaces of the new Objective C implementation. In combination with the detailed description of the functionality in [Bar10] (operations, coordination and transaction details), [Dö11] (runtime, API, configuration details), [Zar12] (persistency layer details) and [63](MozartSpaces Tutorial) it is possible to configure and operate with the new Objective C implementation.

Configuration, startup, and shutdown

The configuration can be done programmatically. The configuration using XML and plist is partially started but not fully implemented right now. The `XMConfiguration` class is similar to the MozartSpaces `Configuration` class and can be used as a parameter for the instantiation of the core (`XMDefaultXaviCore`). It allows configurations like defining embedded space, the request handler or the serializer. Detailed options are described in [Dö11] section 6.3. In listing 6 is sample configuration using an embedded space, defining the MozartSpaces/Objective C serializer (mzsxavi) with a space URL.

The startup is done by creating an `XMDefaultXaviCore` using a configuration and initializing a synchronous API (`XMCapi`). The shutdown of a space is done by defining a space. This is presented in listing 7.

```
XMConfiguration *config = [XMConfiguration configuration];
[config setEmbeddedSpace:YES];
[config setSerializerIds:[NSArray arrayWithObjects:@"mzsxavi", nil]];
NSURL *s = [[NSURL alloc] initWithString:@"xvsm://localhost:9876"];
[config setSpaceUri:s];
```

Listing 6: Configuration of the new Objective C implementation

```
id <XMXaviCore> core = [XMDefaultXaviCore
      defaultXaviCoreWithConfiguration:config];
XMCapi *capi = [XMCapi capiWithCore:core];
[capi shutdownWithSpace:nil Error:&error];
```

Listing 7: Startup and shutdown of the new Objective C implementation

Container operations

The containers are managed by the *container manager* (`XMDefaultContainerManager`). The functionality of the CAPI-3 layer concerning container management is described in detail in [Bar10] section 5.3. The CAPI interface allows creating and destroying as well as looking up of containers. Listing 8 present the creation of a named container, a lookup of a container defined by the container name and the destruction of a container.

```
XMContainerReference *cref =
        [capi createContainerWithName:@"containerName" Error:&error];
XMContainerReference *lookupCref =
        [capi lookupContainerWithName:@"containerName" Error:&error];
[capi destroyContainerWithContainerReference:cref
       TransactionReference:nil Error:&error];
```

Listing 8: Container operations with the new Objective C implementation

Entry operations

Every entry that can be added to a container must be serializable. This means the implementation of the following protocols: `NSCoding, NSObject, NSCopying`. For the XML serializer described in 5.3.1 there exists in addition the marker protocol `XMMzsXaviSerializerMarker` that needs to be implemented or the object must be defined as a primitive data type. Otherwise an `XMSerializationError` occur. The following operations concerning entries are supported: *read*, *write*, *test*, *take*, *delete*. Listing 9 presents the usage of the entry operations. First an entry is inserted in a container. This entry works for the built-in as well as for the XML Serializer. `NSCoding`, `NSObject` and `NSCopying` are implemented by `NSString` and `NSString` is a primitive data type in the XML serializer. All other operations use a default selector choosing one entry. The *read* operation returns an array containing the resulting entries. The *test* operations return the number of entries specified by the selector. The *take* operation returns like the *read* operation the resulting entries but also removes them from the container. The *delete* works like the *take* operation but instead of returning the entries only the number of deleted entries is returned.

```
NSArray *array;
int count;
[capi writeWithEntry:[XMEntry entryWithValue:@"entry"]
       ContainerReference:cref Error:&error];
array = [capi readWithContainerReference:cref Error:&error];
count = [capi testWithContainerReference:cref Error:&error];
array = [capi takeWithContainerReference:cref Error:&error];
count = [capi deleteWithContainerReference:cref Error:&error];
```

Listing 9: Entry operations with the new Objective C implementation

Coordination and selection

All entries of a container are organized by coordinators in a specified way (e.g. FIFO ordering). The corresponding selectors are responsible for selecting the entries in a defined way. For example the FifoCoordinator holds the elements in a first-in first-out ordering that the FifoSelector can access and return to the responding caller. The Stage-Concept allows chaining multiple selectors to process a final result. A detailed description of the coordination concepts is presented in [Bar10] section 7. The following coordinators are implemented right now:

- AnyCoordinator
- FifoCoordinator
- LifoCoordinator
- KeyCoordinator
- LabelCoordinator
- RandomCoordinator

Listing 10 presents the usage of the FifoCoordinator and the corresponding selector. At the beginning a FifoCoordinator is created and registered on container. Then a selector is created that return 3 entries and used for a *read* operation.

```
XMFifoCoordinator *fifoCoordinator = [XMFifoCoordinator
      fifoCoordinator];
NSArray *coordinators = [NSArray arrayWithObject:fifoCoordinator];
NSArray *array;
XMContainerReference *cref =
       [capi createContainerWithSpace:nil
                ObligatoryCoordinators:coordinators
                  TransactionReference:nil
                                 Error:&error];
//write entries
id <XMSelector> selector = [XMFifoCoordinator selectorWithCount:3];
array = [capi readWithContainerReference:cref
                                Selector:selector
                  TimeOutInMilliseconds:
                        XAVI_CONSTANTS_REQUEST_TIMEOUT_DEFAULT
                   TransactionReference:nil Error:&error];
```

Listing 10: Coordination and selection with the new Objective C implementation

Transaction handling

MozartSpaces operations can be grouped into transactional safe operations. A pessimistic concurrency control system using locks is used for the prevention of inconsistent states. The isolation levels `REPEATABLE_READ` and `READ_COMMITTED` are supported and define the level influencing of different transactions. [Bar10] section 6 describes the transaction mechanism in the CAPI-3 layer in detail. The transaction management is done by the transaction manager as described in detail in [Dö11] section 4.2.4. Well-known operations like *commit* and *rollback* are supported as well as timeouts of transactions. The external API `XMTransactionReference` allows transaction handling from the user perspective. Listing 11 presents the principle usage in the new Objective C implementation. An `XMTransactionReference` is created with a timeout of 5,000ms using the embedded space. After some operations the transaction can be committed of roll backed. There is also the possibility to use implicit transactions by using `nil` as transaction reference. Then the runtime creates a transaction that encapsulates only this operation.

```
XMTransactionReference *tx =
        [capi createTransactionWithTimeoutInMilliseconds:5000
                                                    Space:nil
                                                    Error:&error];
//perform some operations using transaction
[capi commitTransactionWithTransactionReference:tx Error:&error];
// or
[capi rollbackTransactionWithTransactionReference:tx Error:&error];
```

Listing 11: Transaction handling with the new Objective C implementation

Aspects

Aspects are used to execute code segments before and after a specific operation. This allows separating the implementation of crosscutting concerns (e.g. logging) from the business logic. There exist aspects that are executed on the whole space (*space aspects*) and aspects that are executed on a specific container (*container aspects*). Aspects are supported with the built-in serializer and is therefore only working in the Objective C environment. The XML serializer does not support aspects. A list of the all space and container aspects is in [63] section 5.2 and 5.3. Listing 12 presents a typical pre-read and post-read implementation of a *container aspect*. The according methods need to be overridden of the `XMAbstractContainerAspect` class. Listing 13 presents the usage the *container aspect*. First is the definition of the interception points (pre-Read, post-Read) and then adding the aspects to the container.

```
@interface SomeAspects : XMAbstractContainerAspect
@end
```

```
@implementation SomeAspects {
}

- (XMAspectResult *)preReadWithRequest:(XMReadEntriesRequest *)
      request
                              Transition:(id <XMTransaction>)tx
                          SubTransaction:(id <XMSubTransaction>)stx
                         Capi3AspectPort:(id <XMCapi3AspectPort>)capi3
                          ExecutionCount:(int)executionCount {
    //preReadAspect tasks
    return [XMAspectResult aspectResultOK];
}

- (XMAspectResult *)postReadWithRequest:(XMReadEntriesRequest *)
      request
                               Transition:(id <XMTransaction>)tx
                           SubTransaction:(id <XMSubTransaction>)stx
                          Capi3AspectPort:(id <XMCapi3AspectPort>)capi3
                           ExecutionCount:(int)executionCount
                                  Entries:(NSArray *)entries {
    //postReadAspect tasks
    return [XMAspectResult aspectResultOK];
}
@end
```

Listing 12: Defining aspects with the new Objective C implementation

```
SomeAspects *aspects = [[SomeAspects alloc] init];
NSSet *ipoints = [NSSet setWithObjects:[XMContainerIPoint cPreRead],
      [XMContainerIPoint cPostRead], nil];

[capi addContainerAspectWithContainerAspect:aspects
        ContainerReference:cref
        IPoints:ipoints
        Error:&error];
[capi readWithContainerReference:cref Error:&error];
```

Listing 13: Using aspects with the new Objective C implementation

Error handling

The following errors are available (the corresponding exceptions are in brackets):

- `XMAspectError` (`AspectException`): Error concerning aspects (e.g. aspect is not registered)
- `XMEntryCopyingError` (`EntryCopyingException`): Error concerning cloning of entries (e.g. request context)

- `XMMetaModelError` (`MetaModelException`): Error concerning meta model (not used at the moment)
- `XMTimeoutError` (`MzsTimeoutException`): Error concerning timeouts (e.g. a task has timed out)
- `XMSerializationError` (`SerializationException`): Error concerning serialization (e.g. serialization error because of missing expected protocol)
- `XMAccessDeniedError` (`AccessDeniedException`): Error concerning authorization framework (not used at the moment)
- `XMContainerFullError` (`ContainerFullException`): Error concerning container operation (e.g. writing to a full container)
- `XMContainerLockedError` (`ContainerLockedException`): Error concerning container locking by transaction
- `XMContainerNameNotAvailableError` (`ContainerNameNotAvailableException`): Container name is already used by another container.
- `XMContainerNotFoundError` (`ContainerNotFoundException`): The container name could not be found.
- `XMCoordinatorLockedError` (`CoordinatorLockedException`): The coordinator is locked by another transaction.
- `XMCoordinatorNotRegisteredError` (`CoordinatorNotRegisteredException`): The coordinator for the used selector cannot be found.
- `XMCountNotMetError` (`CountNotMetException`): The selector count could not be satisfied.
- `XMDuplicateCoordinatorError` (`DuplicateCoordinatorException`): Error when coordinators with duplicate names are registered.
- `XMDuplicateKeyError` (`DuplicateKeyException`): Duplicate key in the KeyCoordinator are used.
- `XMEntryLockedError` (`EntryLockedException`): Error concerning entry locking by transaction.
- `XMEntryNotAnnotatedError` (`EntryNotAnnotatedException`): Error concerning LindaCoordinator (not used at the moment)

- `XMInvalidContainerError` (`InvalidContainerException`): Error when container is invalid.
- `XMInvalidContainerNameError` (`InvalidContainerNameException`): Error when container name is invalid.
- `XMInvalidCoordinatorNameError` (`InvalidCoordinatorNameException`): Error when coordinator name is invalid.
- `XMInvalidEntryError` (`InvalidEntryException`): Error when entry gets invalid.
- `XMInvalidEntryTypeError` (`InvalidEntryTypeException`): Error concerning TypeCoordinator (not used at the moment)
- `XMInvalidSubTransactionError` (`InvalidSubTransactionException`): Error when sub-transaction is invalid.
- `XMInvalidTransactionError` (`InvalidTransactionException`): Error when transaction is invalid.
- `XMInvalidTypeError` (`InvalidTypeException`): Error concerning authorization (not used at the moment)
- `XMObligatoryCoordinatorMissingError` (`ObligatoryCoordinatorMissingException`): Error when no coordinator is associated with a selector.
- `XMPersistenceInitializationError` (`PersistenceInitializationException`): Error concerning persistence layer initialization.

The following exceptions are used in context of unexpected situations:

- `XMAssertionException` (`AssertionError`): Exception for undefined situations in container restoring.
- `XMIllegalArgumentException` (`IllegalArgumentException`): Exception for undefined states if getting illegal arguments.
- `XMIllegalStateException` (`IllegalStateException`): Exception for undefined states.
- `XMMzsCoreRuntimeException` (`MzsCoreRuntimeException`): Unchecked exception in the core.

- `XMNoSuchElementException` (`NoSuchElementException`): Exception if element could not be found in data structure.
- `XMNullPointerException` (`NullPointerException`): Exception if a reference in nil (null).
- `XMPersistenceException` (`PersistenceException`): Exception when operation of the persistence layer fails.
- `XMRemotingException` (`RemotingException`): Exception concerning error in remote communication.
- `XMTransactionException` (`TransactionException`): Exception concerning error in transaction.
- `XMUnsupportedOperationException` (`UnsupportedOperationException`): Exception if operation is not supported at the specified point.

The error handling for the user can be performed as show in listing 14. A transaction will be committed by forwarding a transaction reference and a pointer to an `XMError` object. If an error occurs the error object will be created and can be handled appropriately (e.g. checking for different kind of errors).

```
XMError *error = nil;
[capi commitTransactionWithTransactionReference:tx Error:&error];
if (error != nil) {
    //error handling
    if ([error isKindOfClass:[XMTimeoutError class]]){
    }
}
```

Listing 14: Error handling with the new Objective C implementation

Persistency

Most of the persistency features of MozartSpaces are ported right now except the database related classes. This means that `XMPersistenceBackend` as well as `XMDBAdapter` and the implementing classes for the in-memory storage (`XMInMemoryDB`) are implemented right now. Hence the storage is volatile and all data gets lost after shutdown of the core. The SQLite implementation is partly ported but not runnable at the moment but it might be a good solution because SQLite is part of the iOS environment. Berkeley DB [35] is not included in iOS but can be compiled manually [36].

Getting started!

Listing 15 presents a small example of the usage of the new Objective C implementation to get a starting point for developers. At the beginning the logging facility is initialized. The core is configured, initialized, and is using the synchronous interface `XMCapi`. Then a container using a FifoCoordinator is created and an entry is written. This is encapsulated by a transaction and is committed. After this a selector is configured to read the entry from the container using an implicit transaction. At the end the container is destroyed and the space is shutdown. Error handling is performed by using an error object that needs to be evaluated after every method call outlined in this example by using assertions.

```
[XMLoggerConfiguration configureLogger];

XMError *error = nil;
XMConfiguration *config = [XMConfiguration configuration];
id <XMXaviCore> core = [XMDefaultXaviCore
      defaultXaviCoreWithConfiguration:config];
XMCapi *capi = [XMCapi capiWithCore:core];
NSLog(@"XaviSpaces: transactional 'Getting started!' with synchronous
      interface");

XMTransactionReference *transaction =
        [capi createTransactionWithTimeoutInMilliseconds:5000
                                                   Space:nil
                                                   Error:&error];
NSAssert(error == nil, @"");

NSArray *coords =
        [NSArray arrayWithObjects:
                [XMFifoCoordinator fifoCoordinator], nil];
XMContainerReference *container =
        [capi createContainerWithName:@"helloSpaceContainer"
                                Space:nil
                                 Size:10
                ObligatoryCoordinators:coords
                  OptionalCoordinators:nil
                  TransactionReference:transaction
                                Error:&error];
NSAssert(error == nil, @"");

NSString *entry = @"Hello Space!";
NSArray *entries =
        [NSArray arrayWithObject:
                [XMEntry entryWithValue:entry]];
[capi writeWithEntries:entries
        ContainerReference:container
        TimeOutInMilliseconds:XAVI_CONSTANTS_REQUEST_TIMEOUT_DEFAULT
```

```
        TransactionReference:transaction
        Error:&error];
NSAssert(error == nil, @"");
NSLog(@"Entry written: %@", entry);

[capi commitTransactionWithTransactionReference:transaction
                                          Error:&error];
NSAssert(error == nil, @"");

NSArray *selectors =
        [NSArray arrayWithObject:[XMFifoCoordinator selector]];
NSArray *resultEntries =
        [capi readWithContainerReference:container
                               Selectors:selectors
                  TimeOutInMilliseconds:1000
                   TransactionReference:nil
                                  Error:&error];
NSAssert(error == nil, @"");
NSLog(@"Entry read: %@", [resultEntries objectAtIndex:0]);

[capi destroyContainerWithContainerReference:container
                        TransactionReference:nil Error:&error];
NSAssert(error == nil, @"");

[capi shutdownWithSpace:nil Error:&error];
NSAssert(error == nil, @"");
```

Listing 15: Getting started with the new Objective C implementation

CHAPTER 6

Application Scenario

As an application scenario the well-known "Chat example" of MozartSpaces [47] is used. Multiple users on different peers can send messages to each other and also receive them. The communication between the peers is accomplished via Space based middleware XVSM. The users can join and leave whenever they want - all already sent messages are stored in a single container and will be recovered after connecting and so become visible for the user. The sequence of the messages will be kept in the order they were sent. The new implementation for iOS (iPhone) and for OS X as well as the MozartSpaces implementation is used for the scenario.

The communication between the computer peers is done via LAN connection and over WLAN for the iOS device. The peer that is the first user creates and holds the space (embedded space). When connecting to a remote peer the communication is done using the new XML serializer to get compatibility between Java and Objective C. The message (`MzsXaviMessage` respectively `XMMzsXaviMessage`) that is sent between the peers consists of the username (String) and the text message itself (String). This message is used as entry that is stored in the FIFO container. Therefore it implements `MzsXaviSerializerMarker` (Java) or `XMMzsXaviSerializerMarker` (Objective C).

One peer creates a space by specifying the port number and connects to it with a username. The space contains a list of messages (username and message content) in first-in, first-out order (FIFO). Maintaining the sequence as well as the coordination is accomplished by a FIFOCoordinator. Since the Objective C implementation has no notification mechanism right now a polling thread is implemented for Java and for Objective C to check for new messages on the space. The MozartSpaces implementation of notifications heavily uses aspect and they are not supported at the moment because they cannot be serialized. A polling thread checks every 300ms for updates and reload all messages if there exist new messages on the space. This is a simplification that has big

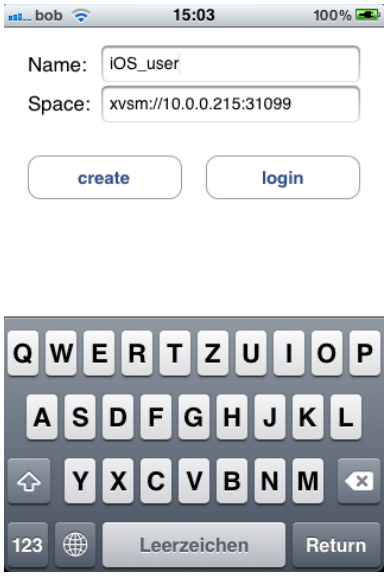

Figure 6.1: iOS chat login window

performance issues but necessary to keep the application as simple as possible. Another user can be connected by specifying the space (IP address and port) and a username. After the connection is established all already existing messages from the space are recovered on the new peer. Figure 6.1 shows the login/create space window.

The following application scenario is used to present the functionality of the new implementation in combination with MozartSpaces: A MozartSpaces client starts as the very first user and creates a space. The core is created with the specified IP address, port and the new XML serializer for remote communication. Then an OS X client connects to the already running space and starts the chat by sending messages between each other. As last peer the iOS (iPhone) implementation joins and receives all already sent messages and lists them in the GUI.

CHAPTER 7

Evaluation

The evaluation of the new Objective C implementation is done on three important criteria: performance, memory usage and compatibility to MozartSpaces. For the serializer the size of the transferred data is benchmarked additionally. For Objective C there are two different configurations used for comparison: Memory management using Garbage Collection (GC) and Automatic Reference Counting (ARC). Although GC is not supported on the iOS environment it gives a feeling what additional future memory management optimizations for ARC bring for performance improvements. At the beginning the new XML serializer will be compared to existing serializers for Java as well as for Objective C. Then the CAPI-3 performance including the scalability will be compared between the existing MozartSpaces implementation and the new implementation. At the end of the performance benchmarks is a comparison of operations on an embedded space. The compatibility to the actual MozartSpaces implementation is another important aspect of the new XVSM implementation for the iOS environment. This is described in the fourth part. At the end of this section is short discussion of the results.

7.1 Benchmark environment

All benchmarks are executed on a MacBook pro (Mid 2009) with 2,53GHz and 8GB RAM. As operating system is MacOS X Lion (10.7) used with the latest updates (July 2013). Java 1.6.0_43 64bit for MacOS is used as JVM. The development environment for Objective C is Xcode 4.6.3 with the Apple LLVM 3.1 compiler.

7.2 Performance benchmark

Because of the lack of equivalent hardware for the iOS environment and the Android environment (there exists no mobile phone that can run Android as well as iOS) the performance benchmark are executed on a Mac for Java and Objective C. Since the implementations for Android and PC as well as the implementation for iOS and Mac have no big differences this design should be a useful indicator for the performance in the iOS environment. The MozartSpaces implementation is used from a snapshot from 4^{th} June, 2012.

All benchmarks are executed ten times and the two fastest and two slowest results are discharged to avoid outliers because of the garbage collector, the just-in-time compiler or other disturbing processes of the operating system. Then the arithmetic mean of the remaining 6 results is calculated and used for comparison. The memory usage is measured using the well-known `top` command for Objective C. The tool returns the memory that the process has allocated from the operating system. The memory usage for Java is calculated by `Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()` with a manual call of the garbage collector if necessary.

7.2.1 Performance benchmark serializer

The serializer has to serialize two objects that cover primitives, collections, and objects containing circular references followed by a deserialization process. The chosen objects simulate messages like `RequestReference` or different kind of `Request` instances transferred between the XVSM implementations. The JavaBuiltin serializer is used as the reference for the other implementations. On the Java environment the new XML serializer and the XStream XML are evaluated. The XStream is also added to get a reference for serialization in XML. On the Objective C environment the new serializer and the built-in serializer (using `NSKeyedArchiver`) are compared. Figure 7.1 gives the result of the performance of the different serializers.

The sums of the size of the byte stream of the two objects are also compared. This is useful because the transmission rate in a mobile environment is sill a limited resource especially using 3G/4G. Figure 7.2 gives the result of the size of the byte stream of the different serializers.

The Objective C built-in serializer using `NSKeyedArchiver` has a performance drawback compared to the Java Builtin. ARC is 3.73 and GC is 4.82 times slower than the corresponding Java implementation. The new XML serializer is not optimized in serialization performance as well as in byte stream output size and is therefore much slower than the Java built-in - ARC by a factor 18.79, GC by a factor 19.91, Java by a factor 34.85. It uses readable code as well as readable output. No redundant information

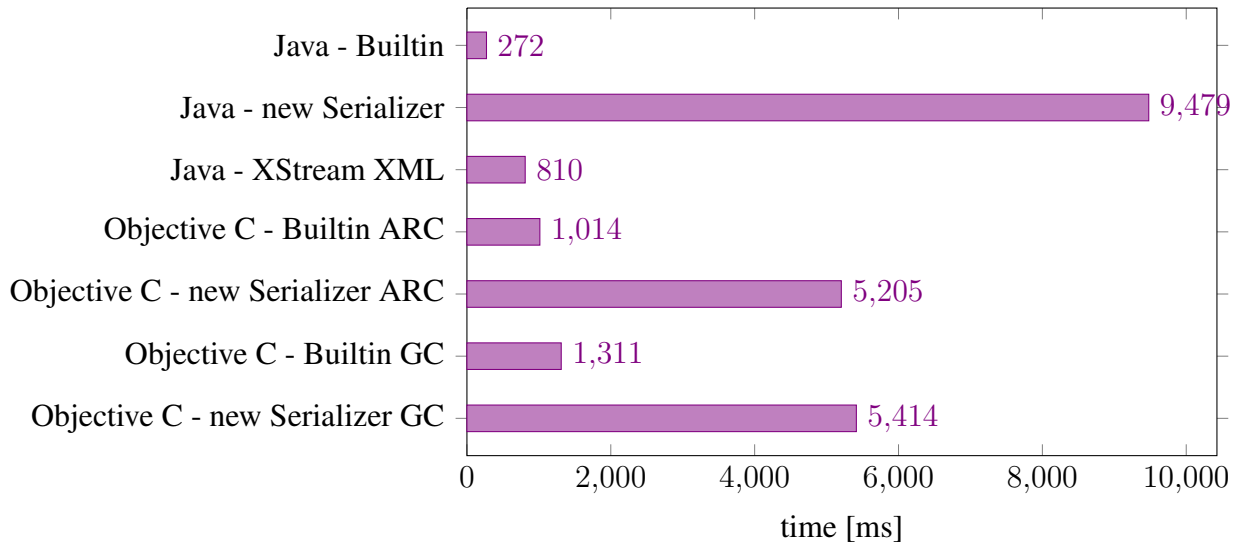

Figure 7.1: Performance evaluation - comparison of different serializers on different platforms using different memory management

is removed to offer optimum readability. As XStream is a mature library it can therefore show a maximum of performance using XML (2.98 times slower than Java built-in).

7.2.2 Performance benchmark CAPI-3

The following two sections describe the benchmarks of the CAPI-3 using the FifoCoordinator and the RandomCoordinator compared to the existing MozartSpaces implementation. They are chosen because they cover the general functionality of the coordinators and the implementation of the other coordinators is quiet similar. While for the LifoCoordinator this is obvious this is also for LabelCoordinator and the KeyCoordinator true because of the usage of HashMaps and therefore also an $O(1)$ time complexity for accessing one entry. While the FifoCoordinator (and the corresponding selector) operates on one element (like LabelCoordinator, KeyCoordinator, LifoCoordinator) the RandomCoordinator operates on the whole dataset and shuffle it. This given a good overview of the performance of frequent operations like “read one entry”, “take all entries” or “read some entries”. The first three benchmark configurations presented in the following sections offer results influenced especially by the implementation of the coordinator while the isolation manager is only little active because of the creating a single transaction per operation. This is relatively cheap in contrast to checking the availability of entries because of entry locking of a transaction. The last three benchmark configurations are much influenced by the isolation manager because it has to check for entry availability. This leads to a dramatic decrease of the performance. All operations are executed

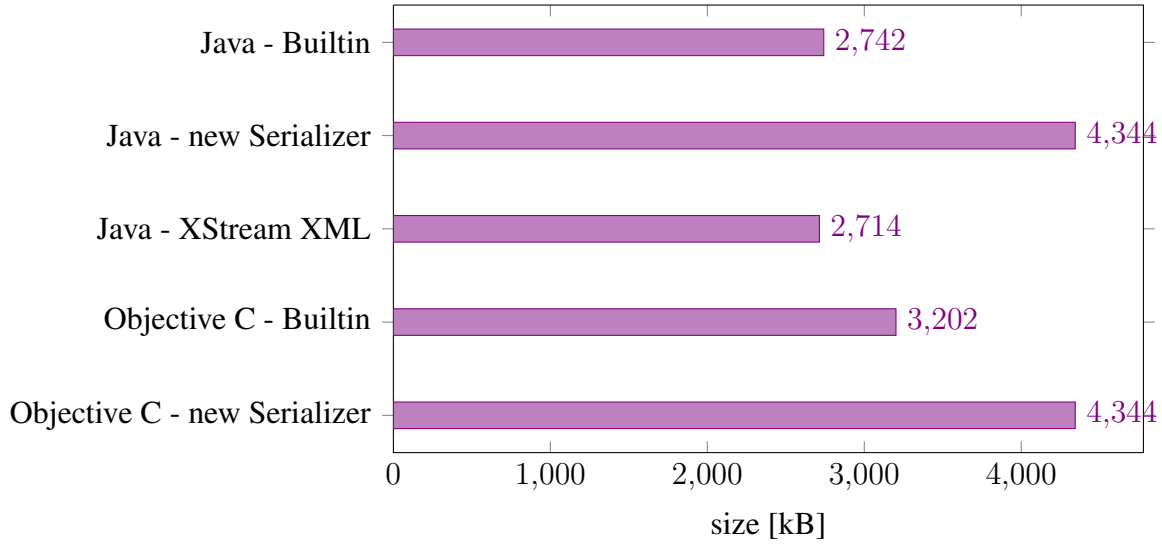

Figure 7.2: Byte stream size evaluation - comparison of different serializer on different platforms

in repeatable-read isolation level. For the read and take operations exist a pre-filled container with exactly the number of entries that where read or taken.

FifoCoordinator

To compare the performance of the new CAPI-3 implementation with the existing one of MozartSpaces to following benchmark configuration is executed on a container using a FifoCoordinator:

- 20,000 read operations using a separate transaction per read (“Read Separate 20,000”)
- 20,000 write operations using a separate transaction per write (“Write Separate 20,000”)
- 20,000 take operations using a separate transaction per take (“Take Separate 20,000”)
- 8,000 read operations using a single transaction (“Read Single 8,000”)
- 20,000 write operations using a single transaction (“Write Single 20,000”)
- 3,000 take operations using a single transaction (“Take Single 3,000”)

Figure 7.3 gives an overview of the FifoCoordinator benchmarks. The new Objective C implementation is from 1.07 (“Read Single”) to 7.63 (“Take Single”) for GC and from

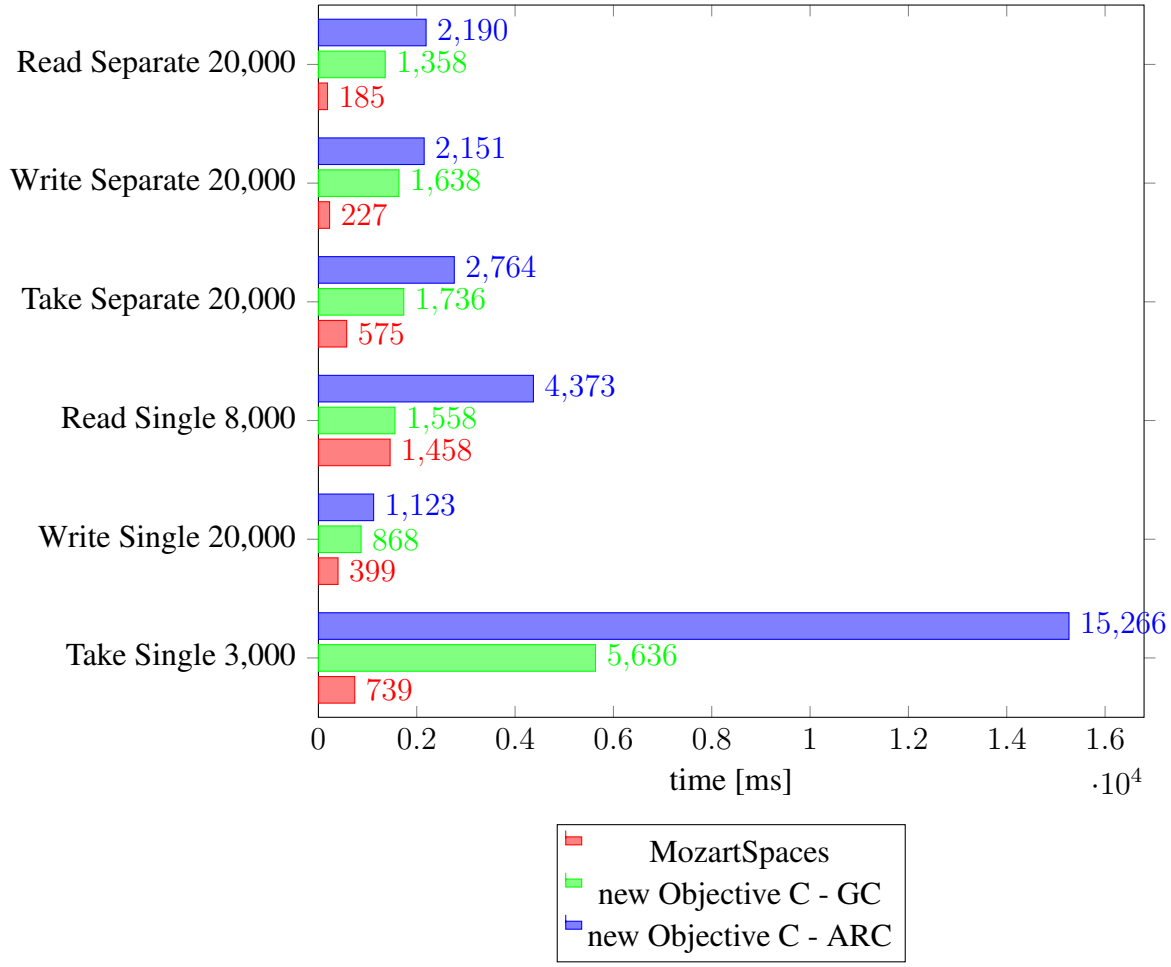

Figure 7.3: Performance evaluation new Objective C compared to MozartSpaces using FifoCoordinator

2.82 (“Write Single”) to 20.67 (“Take Single”) for ARC slower than the corresponding MozartSpaces implementation.

RandomCoordinator

To compare the performance of the new CAPI-3 implementation to the existing one of MozartSpaces following benchmark configuration is executed on a container using a RandomCoordinator:

- 6,000 read operations using a separate transaction per read (“Read Separate 6,000”)
- 20,000 write operations using a separate transaction per write (“Write Separate 20,000”)

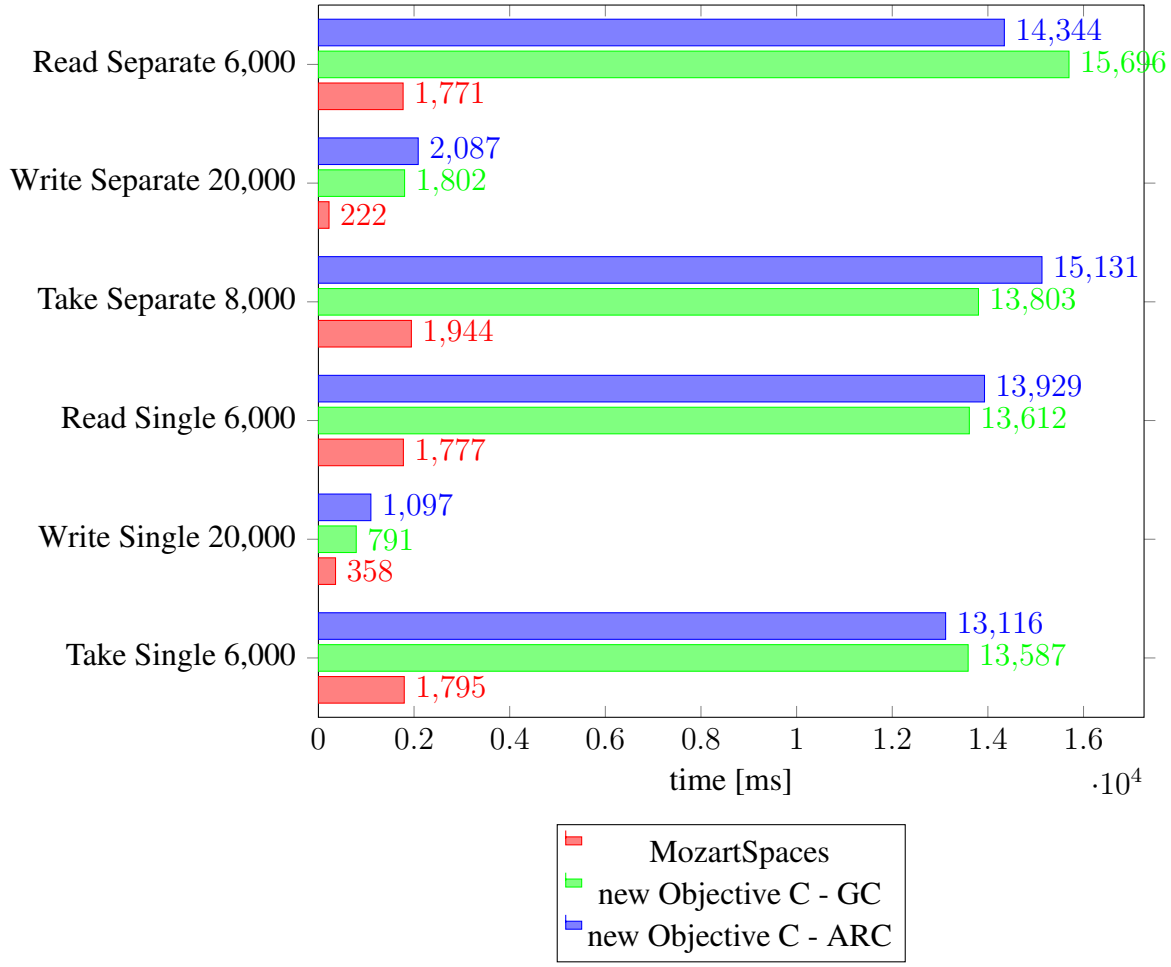

Figure 7.4: Performance evaluation new Objective C compared to MozartSpaces using RandomCoordinator

- 8,000 take operations using a separate transaction per take (“Take Separate 8,000”)
- 6,000 read operations using a single transaction (“Read Single 6,000”)
- 20,000 write operations using a single transaction (“Write Single 20,000”)
- 6,000 take operations using a single transaction (“Take Single 6,000”)

Figure 7.4 gives an overview of the RandomCoordinator benchmarks. The new Objective C implementation is from 2.21 (“Write Single”) to 8.86 (“Read Separate”) for GC and from 3.07 (“Write Single”) to 9.42 (“Write Separate”) for ARC slower than the corresponding MozartSpaces implementation.

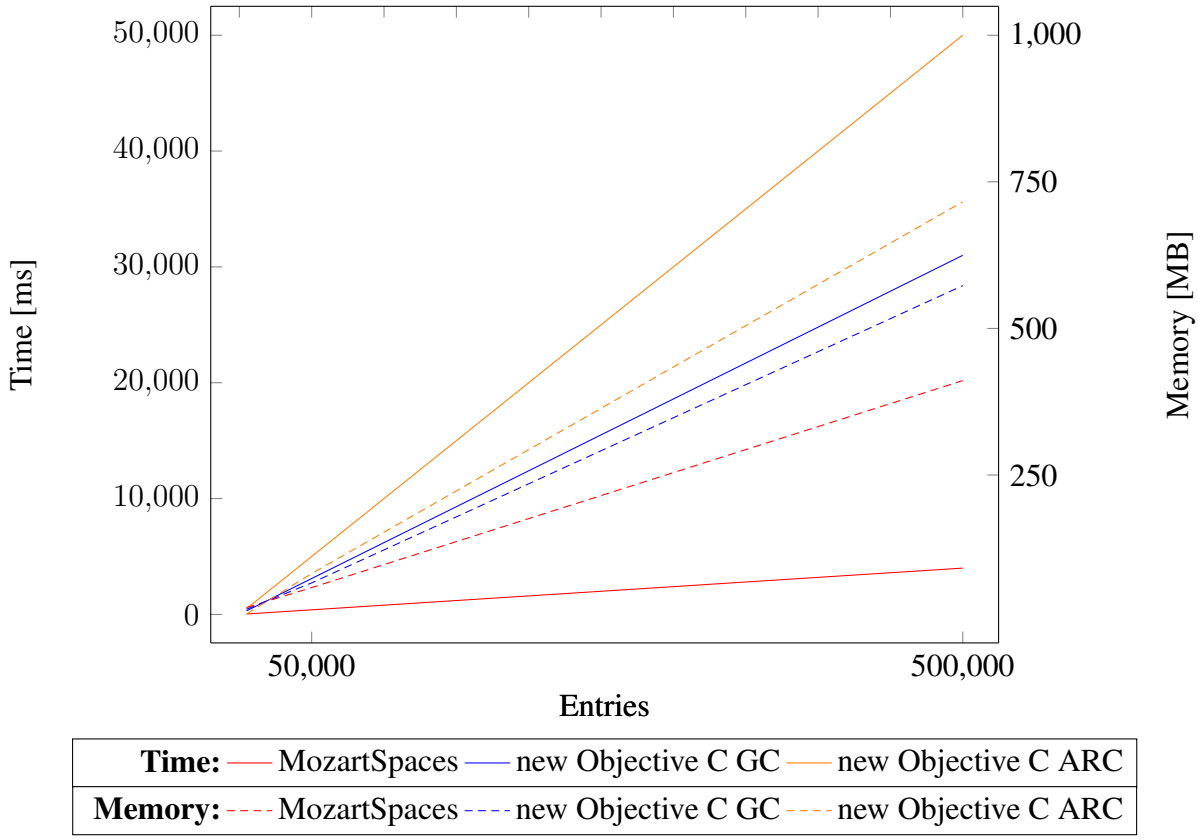

Figure 7.5: Comparison of memory and time scalability of CAPI-3 (FifoCoordinator ("Read Separate Tx"))

7.2.3 Scalability benchmark CAPI-3

Scalability will be evaluated in time as well as in memory consumption. The goal is a linear increase of time and memory dependent on the number of entries. A pre filled container with a FifoCoordinator is used for reading 5,000, 50,000 and 500,000 entries. As shown in Figure 7.5 there is indeed a linear increase of time as well as memory. MozartSpaces is about a factor 12 faster than the new Objective C ARC and about a factor 8 faster than the new Objective C GC. MozartSpaces is the most memory saving implementation. Objective C GC needs about 40% more memory and Objective C ARC needs about 25% more memory.

7.2.4 Performance benchmark embedded space

This section describes the benchmarks of the complete XVSM implementation using an embedded space compared to the existing MozartSpaces implementation. Transport using socket for remote communication is not part of this benchmark. The goal is to get performance values for the runtime environment like messages creation and distribution, requests/responses handling, and tasks. The container uses a FifoCoordinator and all operations are executed using repeatable-read isolation level. The following configuration is used:

- 20,000 read operations using a separate transaction per read ("Read Separate 20,000")
- 20,000 write operations using a separate transaction per write ("Write Separate 20,000")
- 20,000 take operations using a separate transaction per take ("Take Separate 20,000")
- 8,000 read operations using a single transaction ("Read Single 8,000")
- 20,000 write operations using a single transaction ("Write Single 20,000")
- 3,000 take operations using a single transaction ("Take Single 3,000")

Figure 7.6 present the difference between the MozartSpaces and the new Objective C implementation using GC as well as ARC. The new Objective C implementation is from 1.55 ("Read Single") to 5.97 ("Read Separate") for GC and from 3.48 ("Read Single") to 14.72 ("Take Single") for ARC slower than the corresponding MozartSpaces implementation.

Evaluation using embedded space without CAPI-3

Since the configuration of the performance benchmark using embedded space is the same as for the CAPI-3 benchmark using FifoCoordinator the runtime execution time can be calculated. Figure 7.7 present the difference of the execution time of the CAPI-3 benchmark using FifoCoordinator (see figure 7.3) and the execution time of the embedded space using FifoCoordinator (see figure 7.6). This gives the performance of the runtime implementation. The new Objective C implementation is from 1.64 ("Take Single") to 10.91 ("Write Single") times for GC slower than the corresponding MozartSpaces implementation. The ARC implementation is even 0.59 times faster ("Take Single") than MozartSpaces.

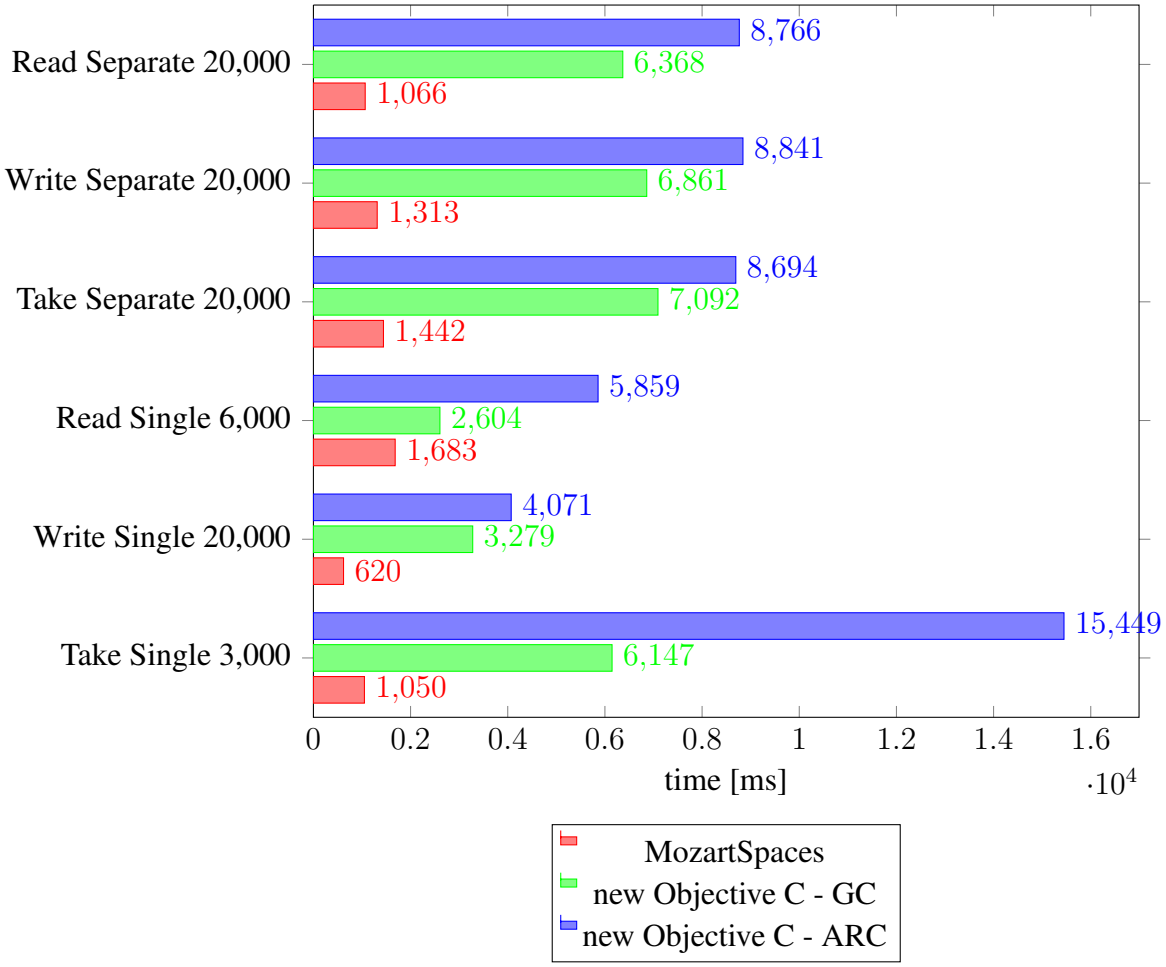

Figure 7.6: Performance evaluation of the new Objective C embedded space compared to MozartSpaces using FifoCoordinator

7.2.5 Scalability benchmark embedded space

Scalability will be evaluated in time as well as in memory consumption. The goal is to show a linear increase of time and memory dependent on the number of entries. Like in the CAPI-3 scalability benchmark 7.5 a pre filled container with a FifoCoordinator is used for reading 5,000, 50,000 and 500,000 entries. As shown in Figure 7.8 there is indeed a linear increase of time as well as memory. MozartSpaces is about a factor 8 faster than the new Objective C ARC and about a factor 5.8 faster than the new Objective C GC. The new Objective C GC implementation is the most memory saving implementation. Objective C ARC needs about 31% more memory, MozartSpaces needs about 30% more memory.

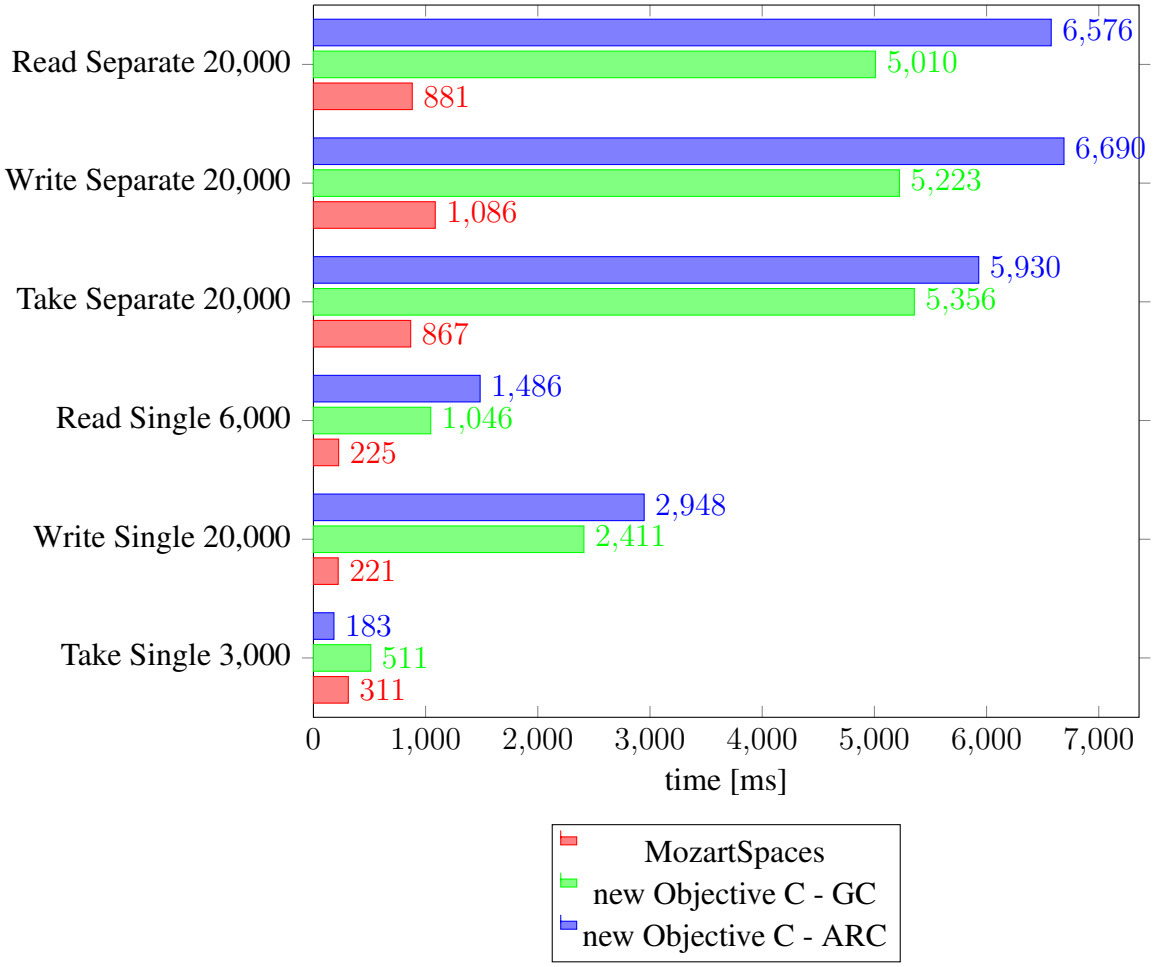

Figure 7.7: Runtime performance evaluation of the new Objective C implementation compared to MozartSpaces

7.3 Compatibility

After comparing the performance the compatibility of the new Objective C implementation and MozartSpaces needs to be evaluated. At first the compatibility is evaluated by using the integration tests as described below. To present the interoperability by example an application scenario is used (see chapter 6).

The code quality of the implementation is evaluated by using unit tests as well as integration tests. While the unit tests take care of the small parts (classes) of the new implementation itself the integration tests check the compatibility between MozartSpaces and the new Objective C implementation. Therefore, integration tests are considered in more detail. The integration tests in MozartSpaces are exactly the same as they are in

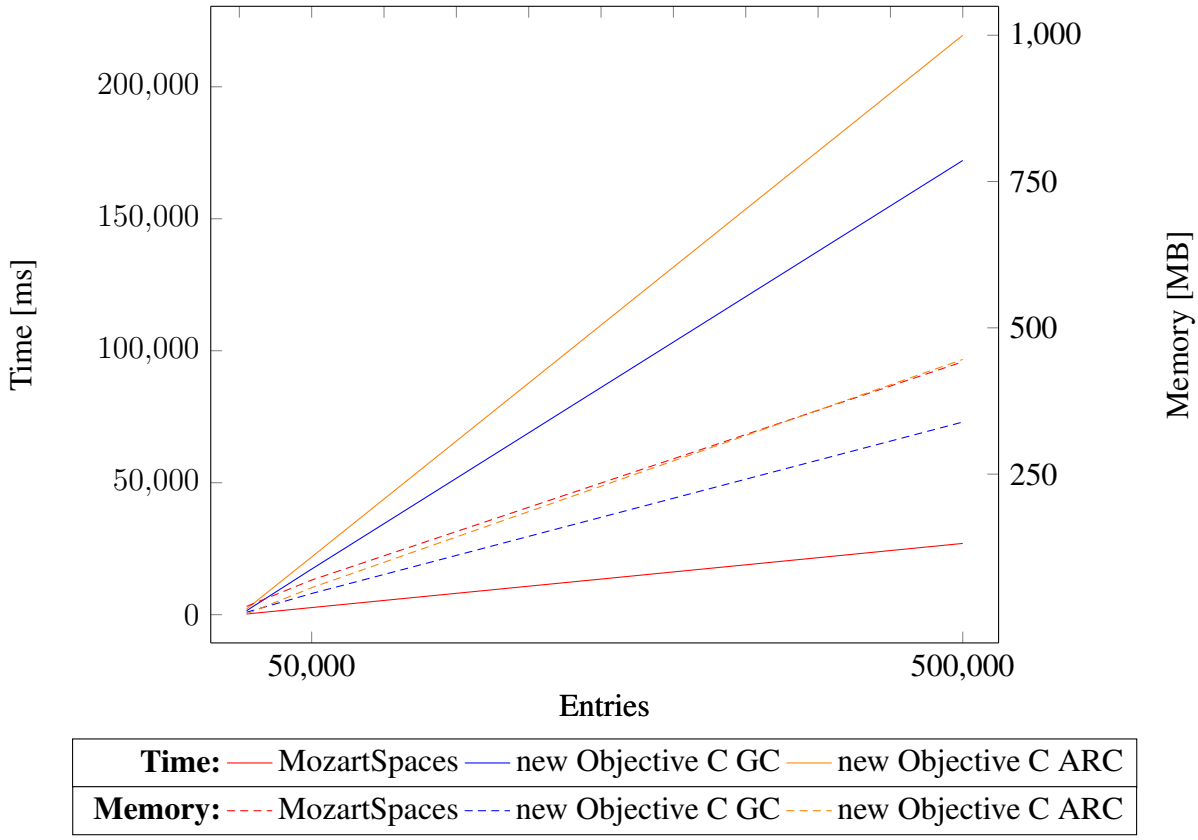

Figure 7.8: Comparison of memory and time scalability of CAPI (FifoCoordinator ("Read Separate Tx"))

the new implementation.

At first all possible objects that can be transmitted between two XVSM instances (e.g. `AnswerContainerInfo`, `ReadEntriesRequest` etc. including their Objective C equivalences `XMAnswerContainerInfo`, `XMReadEntriesRequest` etc.) where created, encoded, and written to a temporary file in the MozartSpaces environment and read, encoded and checked in the Objective C environment. This process is repeated using Objective C environment as source and MozartSpaces as the destination. This method checks if the encoding and decoding in both environments works correctly. It includes especially the compatible serialization and deserialization process in both environments.

Secondly the integration tests are executed. The following configurations of the integration test suite is used to evaluate the new Objective C implementation as well as the compatibility to MozartSpaces:

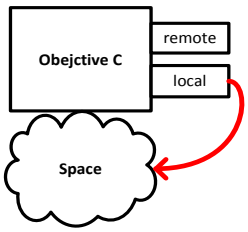

Figure 7.9: Embedded space Objective C

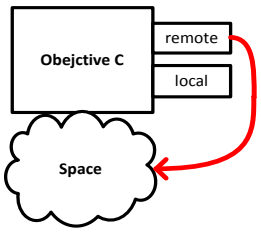

Figure 7.10: Remote space Objective C

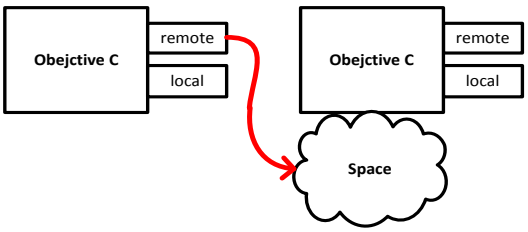

Figure 7.11: Standalone Objective C

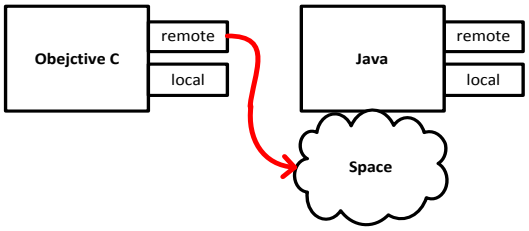

Figure 7.12: Objective C - Java

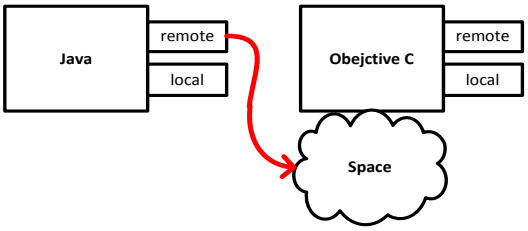

Figure 7.13: Java - Objective C

- Embedded space using the new Objective C implementation (figure 7.9).
- Remote space using space from same process with the new Objective C implementation (figure 7.10).
- Remote space using standalone space with the new Objective C implementation (figure 7.11).
- Remote space of Objective C implementation using standalone space of MozartSpaces (figure 7.12).
- Remote space of MozartSpaces using standalone space of Objective C implementation (figure 7.13).

The red arrows in the figures depict which interface (local or remote) is connected to a space (new Objective C implementation or MozartSpaces(Java)). For example in figure 7.12 the new Objective C implementation access a standalone space of MozartSpaces (Java) using the remote interface (`XMTcpSocketSender` and `XMTcpSocketReceiver`). All integration test scenarios described above are successfully executed. This includes execution of tests with successful operations as well operations that throw an exception/create an `NSError` object.

7.4 Summary and conclusion

The platform independent serialization process is important for the overall performance of future XVSM implementations. The Java Builtin serializer it the fastest method and can therefore works a reference for maximum possible performance. Even the `NSKeyedArchiver` is much slower. Apache Thrift [26] or Protocol Buffers [22] can probably be a good starting point to get high performance platform independent serialization. The overhead of the XML implementation is quiet obvious for both criteria (serialization/deserialization time and byte stream size) and has to be replaced in production systems by a better performing solutions. A performance comparison of different Java serialization methods can be found at [25].

The performance of CAPI-3 is influenced by different aspects. At first is the porting process from MozartSpaces to Objective C. The well performing Java code seems to have some performance issues like using `NSEnumerator` for `Iterator`. Another problem is the heavy usage of small objects like the `Availability` object. Objective C especially in combination with ARC has problems to handle them quickly. Using small C structures can solve that but it has the problem of the manual memory handling because ARC cannot handle C. Another issue is the big difference between GC and ARC. ARC needs to take care of retain and release messages including allocating and freeing memory that seems to work slower than using an extra thread for the garbage collector. Heavy use of `__unsafe_unretained` objects and revising the naming convention can lead to a performance improvement. The next problem is the relatively slow implementation of collections like `NSArray, NSSet,` and `NSDictionary` including their mutable versions. To solve that problem either the usage of the equivalent Core Foundation collections `CFArray, CFSet,` and `CFDictionary` or using C data structures will lead to a real performance boost. The disadvantage would be the manual memory management because ARC does not handle these features.

The time and memory scalability of the CAPI-3 is linear for the MozartSpaces and the new Objective implementation. There is no need for general adaptations because the performance bottleneck is not a design issue but an implementation/programming language problem as described in the upper paragraph.

The runtime performance evaluated by using embedded space configuration comes to the result that the runtime has to wait for CAPI-3 in all “Separate” scenarios as well as in “Read” and “Write” in the “Single” scenario. For the slow CAPI-3 “Take Single” implementation the runtime result has in fact less overhead (183ms for ARC, 511ms for GC and 311ms for MozartSpaces). Investigating the execution using DTrace [38] show that most of the time is spent waiting for releasing a lock. This goes along with the assertion that the performance bottleneck is the CAPI-3 implementation and not the runtime because it has to wait most of the time until the locks are released.

The time and memory scalability of the runtime is linear for MozartSpaces and the new Objective C implementation. As for CAPI-3 there is no need for a general design

adaptation.

An unexpected result is the fact that the Objective C implementation is in some situations clearly slower than the Java implementation. The reason is probably the partially lack of Objective C specific performance optimization. The code seems to work well in Java using the JIT compiler but has performance drawbacks in Objective C. The most surprising result is that the ARC implementation is (sometimes) significantly slower than the GC implementation although Apple presented ARC as a faster technology compared to GC in WWDC 2011 [11]. Due to the facts that iOS only support ARC and GC is set deprecated in OS X 10.8 [9] and will probably be removed in newer versions the future developing should be concentrated on ARC.

The compatibility check using integration tests in different scenarios gives a hint of a code base with a certain quality. Since testing can never verify the correctness but only find errors there is a need of other techniques (e.g. model checking).

All previous performance optimizations where performed using Apple Instruments [10] and DTrace. This will be a good starting point for future performance optimizations for the new Objective C implementation and even for the Java MozartSpaces implementation [37].

It can be summarized that the new Objective C implementation works with an acceptable speed and can interact with MozartSpaces as required in the thesis goals (see 1.1).

CHAPTER 8

Deployment on iOS devices

Developing applications for the iOS environment has some additional tasks that need to be done compared to developing on the Mac. There exits two major differences. At first the development environment and the deploying target is not the same and secondly deploying on Apple iOS devices needs some extra tasks to work. In the beginning of this chapter there is a description of some Apple specific issues like programming restrictions, registration for a developer program and some issues with the App Store and the iOS device simulator. After this a How-To of the deployment process is presented.

8.1 Apple specific issues

8.1.1 Programming restrictions

Apple offers a closed world for their iOS environment. This includes the operating system itself as well as all interfaces defined. The code is closed source and only the public interfaces are allowed to use. Every App gets reviewed concerning interface design, content, functionality and the use of technology. The Apple Review Guidelines [3] give an overview of the restrictions. For example they restrict the used programming language for web access:

> "Apps that browse the web must use the iOS WebKit framework and WebKit JavaScript"

Another restriction is the usage of non-public APIs (e.g. enabling and disabling WLAN interface):

> "Apps that use non-public APIs will be rejected"

For the XVSML development the missing support of long running background processing, missing command line, restricted number of programming languages and restricted API are the main disadvantages. All workarounds presented in 5 uses not allowed techniques. Because of this strict constraint jail-breaking 2.4.2 can outwit this limitations.

8.1.2 Registration

Due to of Apple restrictions all iOS Apps must be signed by a valid certificate before they can be deployed on the iOS device. This needs to be done on the one hand for security reasons and on the other hand to disallow App installation without the App Store [4]. It allows Apple to control the application market (see chapter 8.1.3).

At the beginning there is a need for free registration at the Apple developer portal [5]. This is Apple's information portal concerning developing iOS, OS X and Safari. It gives among others access to the development program and documents like references or sample code. Then there is a need to join an "Apple developer program" for the iOS platform. There can be chosen between 3 different [13]: "iOS Developer Program", "iOS Developer Enterprise Program" and "iOS Developer University Program". The first is for people or organizations that want to submit the Apps to the App Store. The second is designed to for designing proprietary Apps for distribution within an organization. The last one is build for universities and students, free, and is therefore used for this thesis course. One disadvantage is that it does not allow bringing the build App to the AppStore or getting any help of the Apple support. The next step is to create a "Development Certificate". It creates a private/public key pair that allows Xcode to verify the identity of the user. Then the device id (40 digits hex value) must be added to the portal to register a specific developing device. After that an "App ID" must be created. It identifies a set of Apps created by the developing team. Then the "App ID" is added to the "Provisioning Profile". It contains all of the necessary information for Xcode and the magic token that allows deploying on iOS devices. This profile appears in Xcode after login and download in the organizer. Then the connected device must be defined to use for development. After this the device is ready for developing. The whole process is described in detail in [17].

8.1.3 App Store

Since members of the "iOS Developer University Program" cannot submit Apps in the App Store the official link on the Apple website is not accessible. However there exists an unofficial version at another URL[1]. The most important restriction when porting MozartSpaces to the iOS platform is 2.16:

[1] `http://stadium.weblogsinc.com/engadget/files/app-store-guidelines.pdf`

> "Multitasking Apps may only use background services for their intended purposes: VoIP, audio playback, location, task completion, local notifications, etc"

It disallows the background processes and therefore the MozartSpaces implementation ported to iOS will be rejected.

8.1.4 iOS device simulator

Since the compiled App is not runnable on a Mac there is a need of an environment that can be used to run and test the App without a real device. The iPhone/iPad simulator [14] offers this possibility. As part of Xcode it runs like a standard Mac application and can be used to test Apps before deploying on a real device. The simulator can use different iOS versions and different screen resolutions to test the different environments. The interaction can be done with keyboard and mouse to emulate interactions like fingertips, swipes, device rotation or pressing the home button. The simulator appears like a real device with the well-known view of the installed Apps. The interaction works exactly the same as on a real device. Apps can be uninstalled, preferences can be set, and the standard Apple Apps can be used. Debugging in Xcode while using the simulator is also working and is a good starting point for bug fixing. The simulator is a great tool for running new Apps at the beginning but it also has some limitations. The memory is used from the Mac and therefore much bigger than on the real device. Also the user interface performance is different to a real device. There exist also some hardware features that are not simulated like the accelerometer, the camera, the gyroscope, the proximity sensor or the microphone input. Some APIs and frameworks like Apple Push Services or the Event Kit are also missing. Official support for only one simulator instance is another disadvantage. Therefore, the simulator is a good starting point for developing but a real device is necessary to become familiar with the actual environment.

8.2 Deployment How-To

After the registration from chapter 8.1.2 the device is ready for deploying. The following steps are necessary to add the new Objective C implementation to a new Xcode project:

1. Create a static library and the header files depending on the deploying environment. The *Makefile* has different targets for all supported environments.
   `make dist` create all libraries and header files for all environment.
2. Create a new project depending on the environment (iOS or OS X).
3. In **Project Preferences/Build Settings/Other Linker Flages** add
   `-ObjC "/path/to/lib.a"`

4. In **Project Preferences/Build Settings/Header Search Path** add `"/path/to/headers.h"`
5. In **Project Preferences/Build Phases/Link Binary With Libraries** add `CFNetwork.framework`, `Security.framework`, `libsqlite3.dylib`
6. Use chapter 5.3.4 as a starting point for development.

CHAPTER 9

Future Work

The new implementation of a MozartSpaces compatible version of XVSM for the iOS environment seems to work quite well for the first implementation of a complex middleware system in a new programming language as described in chapter 7. But there exist a high number of possible improvements that can make it more and more attractive. The following list presents possible improvements that extend the functionality as well as open issues like programming technique improvements:

- Serialization of aspects is not implemented at the moment for the new XML serializer. This would be necessary to use aspects across heterogeneous platform like MozartSpaces.
- The serialization process is quite slow and the resulting byte stream is big. A new multi platform implementation like Apache Thrift or Protocol Buffer can improve performance heavily. Another option is the usage of an XML schema with the already existing usage of JAXB in MozartSpaces and a new implementation for Objective C.
- The marker protocol `XMMzsXaviSerializerMarker` may not be the right choice to highlight an object that is possible to be serializable with the XML serializer. Either allow serialization of every object or define the restrictions for XML serialization in more detail.
- Although all MozartSpaces integration tests are executed successfully more tests, especially more integration tests including working with test coverage would lead to a quality boost.
- MozartSpaces offers lot of example applications to present inexperienced users a good starting point.

- Some coordinators are missing compared to MozartSpaces like LindaCoordinator, QueryCoordinator or VectorCoordinator.
- The distribution of the new Objective C implementation needs to be defined more clearly. At the moment exist a static library and header files that need to be added to all new projects. In addition there exist other framework dependencies. A framework can make the usage easier.
- The persistence layer is prepared but the implementation for the persistence backend is not implemented right now. SQLite support is partly implemented and probably a good way because SQLite is part if the iOS environment. Berkeley DB [35] is not included in iOS but can be compiled manually. Another possible solution is the usage of Apple's Core Data [8] as persistency backend.
- All authorization features from MozartSpaces are not available and needs implementation.
- Using the space in the 3G environments is still open topic that needs to be solved.
- Mobile devices restrictions like running out of memory needs to be handled to persist entries before the application gets killed.
- Adaptation of the implementation for ARC to safe memory (e.g. more use of AutoreleasePools or changing method name to fulfill the naming convention that result in fewer retain/release calls) and allow faster execution needs to be done.
- The data type compatibility between the iOS environment, Mac and Java needs to be reviewed. The different length of data types depending on the device in Objective C needs a clear definition.
- Because of performance reasons some parts of the implementation like collections, enumerators or intensive use of small objects needs adaptation of the Objective C code or switch to C source code.
- The porting process is based on June 4^{th}, 2012 and therefore all bug fixes as well as other improvements are missing in the actual Objective C implementation.
- A well-defined performance benchmark environment for Objective C (as well as for MozartSpaces) could be used to tweak the implementations.
- The iOS environment offers features like GCD to parallelize operations instead of using thread and replacing locks. If still using locks the `@synchronized` can maybe replaced by POSIX mutex.

- Some not necessary but well performing features like `NonPollingTimeoutProcessor` are not implemented yet.
- At the moment the error handling is split into two different types: error objects for common errors like “container locked” and exceptions for unexpected errors like “wrong list index”. Because exception handling is slow in Objective C a change to error objects may give a performance boost.
- MozartSpaces features like Notifications are not implemented yet.
- The implementation is working for at least iOS 5. Since some useful features (e.g. `NSMapTable`) are available for iOS 6 an update would be worth. According to WWDC 2013 announcement iOS 7 (release date: fall 2013, Beta for developer is available) will support background processing. Therefore, this update sounds very promising.
- An environment for automation of test executions should be created. Since there exist lots of tests that should be performed after every change a continuous integration system like [39] can help saving a lot of time and improve code quality.
- The configuration can be via XML or plist is not fully implemented yet. This feature will provide configuration without recompiling.
- The meta model functionality is not implemented yet.
- The logging framework CocoaLumberjack has some features that extend the functionality dramatically like dynamic changing of the logging level or logging to files.
- The *Makefile* needs revision to get a clear command line interface.
- The actual implementation is compiled for the ARMv7 instruction set but not for the ARMv7s used by the Apple A6 chip (iPhone 5).

CHAPTER 10

Conclusion

The main goal of this thesis was to provide an iOS implementation that is fully compatible to actual reference implementation of XVSM, MozartSpaces. Different approaches had to be mentioned and the most promising solution should be realized.

The goal was achieved by investigating different approaches for code reuse and then taken the decision to port the well performing Java implementation to the iOS environment. The communication between MozartSpaces and the new Objective C implementation was realized by using a simple XML serializer to solve the encoding problem between these two different environments. Some restrictions concerning the iOS environment were also discussed.

The iOS implementation is fully compatible to MozartSpaces. It includes the API definition, most parts of the implementation details as well as the communication process. All well known operations (read, write, take, test) have the same semantics as MozartSpaces. Transaction support with different isolation levels (read committed, repeatable read) are also working as well as different coordination mechanisms (AnyCoordinator, FifoCoordinator, LifoCoordinator, KeyCoordinator, LabelCoordinator, RandomCoordinator). The runtime functionality and the aspect semantic work like in MozartSpaces. The communication based on XML was realized by mapping Java data types to corresponding Objective C data types including the different message types.

The implementation was compared to MozartSpaces by benchmarking concerning execution speed as well as memory usage. The new Objective C implementation was at maximum a factor 15 slower than MozartSpaces. Most operations were about a factor six to eight slower. In one situation (CAPI-3) was MozartSpaces more memory efficient, in the runtime benchmark was the new Objective C implementation better. The compatibility was evaluated but using all integration tests for the new Objective C implementation and presented by a short application scenario (chat example).

The limiting scope of a diploma thesis led to some compromises to be able to perform the proof of concept. The policy was to fully implement the API but leave out detailed implementations. Some coordinators (VectorCoordinator, LindaCoordinator, QueryCoordinator) or the configuration via XML files were not implemented. The usage of aspects in the heterogeneous environment was also skipped.

As described in 2.4.1 there exist different approaches for multi-platform development (i.e. maintaining one code base for PC/Android OS/iOS/etc). This approach can be taken into account for future implementations to save developing time as well as improving code quality because each functionality must be implemented once. Probably, an adaptation of the license model or the use of a commercial software system is needed to reach the goal.

APPENDIX A

Appendix

A.1 Source code heavily used in MozartSpaces

```
package at.TypicalJava;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.atomic.AtomicLong;

/**
 * @author gerry
 *
 *          This class contains typical Java source code that is used
        in
 *          MozartSpaces
 *
 */
public class TypicalJava implements ITypicalJava {
        private static final long serialVersionUID = 1L;
        private static final AtomicLong SOMEATOMICLONG = new
                AtomicLong();
        private List<String> concurrentList;
        private volatile boolean someVolatile;
        private long someLong;

        public TypicalJava() {
                this.someLong = SOMEATOMICLONG.incrementAndGet();
                this.concurrentList = Collections.synchronizedList(
                        new ArrayList<String>());
                this.someVolatile = true;
```

```
        }

        public void doSomeThing() {
                doSomeThingOther();
        }

        private void doSomeThingOther() {
                this.someLong = SOMEATOMICLONG.incrementAndGet();
                this.someVolatile = false;
        }

        public List<String> getConcurrentList() {
                return concurrentList;
        }

        public boolean isSomeVolatile() {
                return someVolatile;
        }

        public long getSomeLong() {
                return someLong;
        }

}
```

Listing 16: Typical Java source code in MozartSpaces

A.2 Java2objc Objective C output

```
#import "NSMutableArray.h"
#import "Collections.h"
#import "NSMutableArray.h"
#import "AtomicLong.h"

/**
 * @author gerry
 *
 * This class contains typical Java source code that is used in
 * MozartSpaces
 *
 */

@interface TypicalJava : NSObject <ITypicalJava> {
  NSMutableArray * concurrentList;
  BOOL someVolatile;
  long someLong;
}
```

```
@property(nonatomic, retain, readonly) NSMutableArray *
      concurrentList;
@property(nonatomic, readonly) BOOL someVolatile;
@property(nonatomic, readonly) long someLong;
- (id) init;
- (void) doSomeThing;
@end
```

Listing 17: Typical Java source code in MozartSpaces converted by java2objc - interface file

```
#import "TypicalJava.h"

long const serialVersionUID = 1L;
AtomicLong * const SOMEATOMICLONG = [[[AtomicLong alloc] init]
      autorelease];

@implementation TypicalJava

@synthesize concurrentList;
@synthesize someVolatile;
@synthesize someLong;

- (id) init {
  if (self = [super init]) {
    someLong = [SOMEATOMICLONG incrementAndGet];
    concurrentList = [Collections synchronizedList:[[[NSMutableArray
          alloc] init] autorelease]];
    someVolatile = YES;
  }
  return self;
}

- (void) doSomeThing {
  [self doSomeThingOther];
}

- (void) doSomeThingOther {
  someLong = [SOMEATOMICLONG incrementAndGet];
  someVolatile = NO;
}

- (void) dealloc {
  [concurrentList release];
  [super dealloc];
}

@end
```

Listing 18: Typical Java source code in MozartSpaces converted by java2obj - implementation file

A.3 Makefile

```
#    Makefile
#    XaviSpaces
#
#        Makefile for executing tests on MacOS X platform and iOS
       simulator platform
#
#        TEST ... test to run
#        GHUNIT_CLI=1 ... run tests
#        GHUNIT_SUITE=1 ... run test suite
#        GHUNIT_SUITE=0 ... run all tests
#        GHUNIT_TEST_GUI=1 ... run tests with gui
#        GHUNIT_TEST_GUI=0 ... run tests on command line
#        GHUNIT_SUITE_TESTS= ... comma separated list of test classes
       (receives classes from command line)
#        GHUNIT_STANDALONE=1 ... run a standalone space
#
#        GHUNIT_INTEGRATION_TYPE=
#        TYPE ... either OBJC_JAVA or JAVA_OBJC
#        OBJECT ... Objective C class name
#
#        examples:
#                make testiOSSimulation #run all tests in simulation
       environment on command line
#                make testiOSSimulation TEST=TestIOS #run specified
       test in simulation environment on command line
#                make testMacAllCli
#                make testMacSuiteCli TESTS=TestMac,TestMisc
#                make testMacAllGui
#                make testMacFile FILE=testsToRunMac
#                make testMacDebug TEST=TestSomething
#
#    created by gerry

default: all
        # Set default make action here
        # xcodebuild -target Tests -configuration MyMainTarget -sdk
              iphonesimulator build

clean:
        xcodebuild -configuration Debug clean
```

```
        xcodebuild -configuration Release clean
        -rm -rf build/*

####################

all:
        xcodebuild -alltargets

####################

list:
        xcodebuild -list

sdk:
        xcodebuild -showsdks

xcode:
        xcode-select -print-path
        xcodebuild -version

####################

dist: distDS51 distRS51 distDS61 distRS61 distDi61 distRi61 distDM107
       distRM107


### (Debug/Release) (Simulator/iOS/Mac) (Version)
### Simulator
distDS51:
        xcodebuild -target XaviSpacesLibiOS -configuration Debug -sdk
                iphonesimulator5.1 build
        mv build/Debug-iphonesimulator build/Debug-iphonesimulator5.1

distRS51:
        xcodebuild -target XaviSpacesLibiOS -configuration Release -
               sdk iphonesimulator5.1 build
        mv build/Release-iphonesimulator build/Release-
               iphonesimulator5.1

distDS61:
        xcodebuild -target XaviSpacesLibiOS -configuration Debug -sdk
                iphonesimulator6.1 build
        mv build/Debug-iphonesimulator build/Debug-iphonesimulator6.1

distRS61:
        xcodebuild -target XaviSpacesLibiOS -configuration Release -
               sdk iphonesimulator6.1 build
        mv build/Release-iphonesimulator build/Release-
               iphonesimulator6.1
```

```
### iOS
distDi61:
	xcodebuild -target XaviSpacesLibiOS -configuration Debug -sdk
		iphoneos6.1 build
	mv build/Debug-iphoneos build/Debug-iphoneos6.1

distRi61:
	xcodebuild -target XaviSpacesLibiOS -configuration Release -
		sdk iphoneos6.1 build
	mv build/Release-iphoneos build/Release-iphoneos6.1

### OS X
distDM107:
	xcodebuild -target XaviSpacesLibMac -configuration Debug -sdk
		macosx10.7 build
	mv build/Debug build/Debug-osx10.7

distRM107:
	xcodebuild -target XaviSpacesLibMac -configuration Release -
		sdk macosx10.7 build
	mv build/Release build/Release-osx10.7



####################
testMacStandalone:
	#xcodebuild -target TestingMac -scheme TestingMac_Standalone
		-configuration Debug -sdk macosx build
	xcodebuild -target TestingMac -configuration Debug -sdk
		macosx build
	DYLD_FRAMEWORK_PATH=./Framework DYLD_FRAMEWORK_PATH=./
		Framework GHUNIT_STANDALONE=1 build/Debug/TestingMac.
		app/Contents/MacOS/TestingMac

testMacIntegration:
	GHUNIT_INTEGRATION=1 GHUNIT_CLI=1 GHUNIT_INTEGRATION_TYPE=${
		TYPE} GHUNIT_INTEGRATION_OBJECT=${OBJECT} xcodebuild -
		target TestingMac -configuration Debug -sdk macosx
		build

testMacSuiteCli:
	GHUNIT_CLI=1 GHUNIT_SUITE=1 GHUNIT_TEST_GUI=0
		GHUNIT_SUITE_TESTS=${TESTS} xcodebuild -target
		TestingMac -configuration Debug -sdk macosx build

testMacFile:
	GHUNIT_CLI=0 GHUNIT_SUITE=1 GHUNIT_TEST_GUI=0
		GHUNIT_SUITE_FILE_TESTS=${FILE} xcodebuild -target
```

```
                TestingMac -configuration Debug -sdk macosx build

testMacAllCli:
        GHUNIT_CLI=1 GHUNIT_SUITE=0 GHUNIT_TEST_GUI=0 xcodebuild -
                target TestingMac -configuration Debug -sdk macosx
                build

testMacAllGui:
        GHUNIT_CLI=1 GHUNIT_SUITE=0 GHUNIT_TEST_GUI=1 xcodebuild -
                target TestingMac -configuration Debug -sdk macosx
                build

testMacDebug:
        GHUNIT_CLI=1 xcodebuild -target TestingMac -configuration
                Debug -sdk macosx build

####################

testiOSSimulation:
        GHUNIT_CLI=1 GHUNIT_SUITE=0 GHUNIT_TEST_GUI=0
                CFFIXED_USER_HOME=/tmp xcodebuild -target TestingiOS -
                configuration Debug -sdk iphonesimulator build

testiOSDebug:
        GHUNIT_CLI=1 CFFIXED_USER_HOME=/tmp xcodebuild -target
                TestingiOS -configuration Debug -sdk iphonesimulator
                build

testiOSDevice:
        GHUNIT_CLI=1 CFFIXED_USER_HOME=/tmp xcodebuild -target
                TestingiOS -configuration Debug -sdk iphonesimulator
                build


testI:
        GHUNIT_CLI=1 CFFIXED_USER_HOME=/tmp xcodebuild -target
                TestingiOS -configuration Debug -sdk iphonesimulator
                build

####################

testAll: testMacAllCli testiOSAllSimulator

testSuite: testMacSuite testiOSSuiteSimulator

####################

benchmarkMac:
```

```
        xcodebuild -target BenchmarkingMac -configuration Release -
                sdk macosx build
        build/Release/BenchmarkingMac

benchmarkiOS:
        xcodebuild -target BenchmarkingiOS -configuration Release -
                sdk iphonesimulator build
```

Listing 19: Makefile

References

[Bar10] Barisits, Martin-Stefan. Design and Implementation of the next Generation XVSM Framework - Operations, Coordination and Transaction. Master's thesis, Vienna University of Technology, 2010.

[Brü13] Andreas Brückl. Relaxed non-blocking distributed transactions for the eXtensible virtual shared memory. Master's thesis, Vienna University of Technology, 2013.

[Buc10] J Bucanek. *Learn Objective-C for Java Developers*. Learn Series. Apress, 2010.

[CDJ+13] Stefan Craß, Tobias Dönz, Gerson Joskowicz, eva Kühn, and Alexander Marek. Securing a space-based service architecture with coordination-driven access control. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 4(1):76–97, 3 2013.

[CDK05] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design (4th Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[CFL+06] Giacomo Cabri, Luca Ferrari, Letizia Leonardi, Marco Mamei, Franco Zambonelli, and Università Di Modena E. Uncoupling coordination: Tuple-based models for mobility. In *The Handbook of Mobile Middleware*. Auerbach Publications, 2006.

[Cra09] Craß, Stefan and Kühn, Eva and Salzer, Gernot. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, number 6 in IDEAS '09, pages 301–306, New York, NY, USA, 2009. ACM.

[Cra10] Craß, Stefan. A Formal Model of the Extensible Virtual Memory (XVSM) and its implementation in Haskell - Design and Specification. Master's thesis, Vienna University of Technology, 2010.

[Dö11] Dönz, Tobias. Design and Implementation of the next Generation XVSM Framework - Runtime, Protocol and API. Master's thesis, Vienna University of Technology, 2011.

[eKMKS09] eva Kühn, Richard Mordinyi, Laszlo Keszthelyi, and Christian Schreiber. Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '09, pages 625–632, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[eKMS08] eva Kühn, Richard Mordinyi, and Christian Schreiber. An extensible space-based coordination approach for modeling complex patterns in large systems. In *ISoLA*, pages 634–648. Springer-Verlag, 2008.

[eKMS+12] eva Kühn, Alexander Marek, Thomas Scheller, Vesna Sesum-Cavic, Michael Vögler, and Stefan Craß. A space-based generic pattern for self-initiative load clustering agents. In *COORDINATION*, pages 230–244, 2012.

[Floon] Florian Lukschander. Thesis on eXtensible Virtual Shared Memory on the Android Operating System. Master's thesis, Vienna University of Technology, In preparation.

[FRL09] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. *ACM Trans. Auton. Adapt. Syst.*, 4(3):16:1–16:26, jul 2009.

[Gel85] Gelernter, David. Generatice communication in Linda. *ACM Transactions on Programming Languages and Systems*, pages 7(1):80–112, 1985.

[Hir12] Jürgen Hirsch. An adaptive and flexible replication mechanism for mozartspaces, the xvsm reference implementation. Master's thesis, Vienna University of Technology, 2012.

[HJR+03] Marjan Hericko, Matjaz B. Juric, Ivan Rozman, Simon Beloglavec, and Ales Zivkovic. Object serialization analysis and comparison in Java and .NET. *SIGPLAN Not.*, 38(8):44–54, aug 2003.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[JYYL09] Liu Jingyong, Zhong Yong, Chen Yong, and Zhang Lichen. Middleware-based distributed systems software process. In *Proceedings of the 2009*

International Conference on Hybrid Information Technology, ICHIT '09, pages 345–348, New York, NY, USA, 2009. ACM.

[Mar09] Markus Karolus. Design and Implementation of XcoSpaces, the .Net Reference Implementation of XVSM – Coordination, Transactions and Communication. Master's thesis, Vienna University of Technology, 2009.

[Mar10] Marek, Alexander. Design and Implementation of TinySpaces, the .NET Micro Framework based implementation of XVSM for embedded systems. Master's thesis, Vienna University of Technology, 2010.

[Mor10] Richard Mordinyi. *Managing complex and dynamic software systems with space-based computing*. PhD thesis, Vienna University of Technology, 2010.

[SB03] Douglas C. Schmidt and Frank Buschmann. Patterns, frameworks, and middleware: Their synergistic relationships. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 694–704, Washington, DC, USA, 2003. IEEE Computer Society.

[SM12] Audie Sumaray and S. Kami Makki. A comparison of data serialization formats for optimal efficiency on a mobile platform. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, number 6 in ICUIMC '12, pages 48:1–48:6, New York, NY, USA, 2012. ACM.

[SS02] Richard E. Schantz and Douglas C. Schmidt. Middleware for distributed systems - evolving the common structure for network-centric applications. In *Encyclopedia of Software Engineering (J. Marciniak and G. Telecki, eds.)*, New York, 2002. Wiley & Sons.

[Tho08] Thomas Scheller. Design and Implementation of XcoSpaces, the .Net Reference Implementation of XVSM – Core Architecture and Aspects. Master's thesis, Vienna University of Technology, 2008.

[TS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., 2006.

[Zar12] Zarnikov, Jan. Energy-efficient Persistence for Extensible Virtual Shared Memory on the Android Operating System. Master's thesis, Vienna University of Technology, 2012.

Web References

[1] Statista (2012): Worldwide market share forecast of smartphone operating systems from 2010 to 2015 from Gartner. http://www.statista.com/statistics/150842/market-share-forecast-of-smartphone-operating-systems\-from-2010-to-2015/.

[2] Statista (2012): Worldwide smartphone shipments from 2010 to 2016 (in million units) from IDC. http://www.statista.com/statistics/12865/forecast-for-sales-of-smartphones-worldwide/.

[3] Apple Inc. App review guidelines. https://developer.apple.com/appstore/guidelines.html.

[4] Apple Inc. Apple app store for ios. http://itunes.apple.com/en/genre/ios/id36?mt=8.

[5] Apple Inc. Apple developer portal. https://developer.apple.com.

[6] Apple Inc. Archives and serializations programming guide. http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Archiving/Archiving.pdf.

[7] Apple Inc. Coding guidelines. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.pdf.

[8] Apple Inc. Core data programming guide. http://developer.apple.com/library/mac/documentation/cocoa/Conceptual/CoreData/CoreData.pdf.

[9] Apple Inc. Garbage collection programming guide. https://developer.apple.com/legacy/library/documentation/Cocoa/Conceptual/GarbageCollection/GarbageCollection.pdf.

[10] Apple Inc. Instruments user guide. http://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/InstrumentsUserGuide.pdf.

[11] Apple Inc. Introducing automatic reference counting. http://adcdownload.apple.com//wwdc_2011/adc_on_itunes__wwdc11_sessions__pdf/323_intro_to_arc_304repeat.pdf.

[12] Apple Inc. ios app programming guide. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/iPhoneAppProgrammingGuide.pdf.

[13] Apple Inc. ios developer programs. https://developer.apple.com/programs/start/ios/.

[14] Apple Inc. ios simulator user guide. http://developer.apple.com/library/ios/documentation/IDEs/Conceptual/iOS_Simulator_Guide/iOS_Simulator_Guide.pdf.

[15] Apple Inc. Memory management. http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/MemoryMgmt.pdf.

[16] Apple Inc. Synchronization. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Multithreading/Multithreading.pdf.

[17] Apple Inc. Tools workflow guide for ios. http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/ios_development_workflow/iphone_development.pdf.

[18] Canalys. Statista (2012): Worldwide market share of leading smartphone OS vendors from 3rd quarter 2010 to 2nd quarter 2012 from Canalys. http://www.statista.com/statistics/172562/worldwide-market-share-of-leading-smartphone-operating\-systems/.

[19] code.google.com. in-the-box. http://code.google.com/p/in-the-box/.

[20] code.google.com. J2objc. http://code.google.com/p/j2objc/.

[21] code.google.com. java2objc. http://code.google.com/p/java2objc.

[22] code.google.com. Protocol buffers. http://code.google.com/p/protobuf/.

[23] Department of Computer Science University of Illinois. Arc specification. http://clang.llvm.org/docs/AutomaticReferenceCounting.html.

[24] dpa. Jailbreak permission in germany. http://www.macwelt.de/news/Freiheit-fuers-iPhone-Anwaeltin-Jailbreak-auch-in-\Deutschland-legal-3205800.html?redirect_seitennr=1.

[25] eishay. Benchmark serialization. https://github.com/eishay/jvm-serializers/wiki.

[26] Apache Software Foundation. Apache thrift. http://thrift.apache.org.

[27] Jay Freeman. Cydia. http://cydia.saurik.com.

[28] Gabriel Handford. Ghunit. https://github.com/gabriel/gh-unit.

[29] GitHub. Fmdatabase. https://github.com/ccgus/fmdb.

[30] GitHub. Gcdasyncsocket. https://github.com/robbiehanson/CocoaAsyncSocket.

[31] GitHub. Ochamcrest. https://github.com/hamcrest/OCHamcrest.

[32] GitHub. Ocmockito. https://github.com/jonreid/OCMockito.

[33] Google Inc. Google play. https://play.google.com/store?hl=en.

[34] GigaSpaces Technologies Inc. GigaSpaces. http://www.gigaspaces.com.

[35] Oracle inc. Berkeley db. http://www.oracle.com/technetwork/products/berkeleydb.

[36] Oracle inc. Berkeley db installation. http://docs.oracle.com/cd/E17076_02/html/installation/index.html.

[37] Oracle inc. Dtrace probes in hotspot vm. http://docs.oracle.com/javase/6/docs/technotes/guides/vm/dtrace.html.

[38] Oracle inc. Solaris dynamic tracing guide. http://docs.oracle.com/cd/E19253-01/817-6223/.

[39] Jenkins community. Jenkins - an extendable open source continuous integration server. http://jenkins-ci.org.

[40] junit team. Junit. http://junit.org.

[41] Igor Khomenko. Stun-ios. https://github.com/soulfly/STUN-iOS.

[42] Mulle kybernetiK. Ocmock. http://ocmock.org/.

[43] U.S. Copyright Office. Jailbreak permission in the usa. http://www.copyright.gov/fedreg/2010/75fr43825.pdf.

[44] Monobjc Project. Monobjc. http://www.monobjc.net.

[45] Arno Puder, Sascha Häberling, Wolfgang Korn, et al. Xmlvm. http://xmlvm.org.

[46] Robbie Hanson. Cocoalumberjack. https://github.com/robbiehanson/CocoaLumberjack.

[47] Space Based Computing Group. Mozartspaces. http://www.mozartspaces.org.

[48] Space Based Computing Group. Space based computing. http://www.spacebasedcomputing.org.

[49] Space Based Computing Group. XVSM. http://www.xvsm.org.

[50] SQLite-Team. Sqlite. http://www.sqlite.org.

[51] W3C. Extensible markup language (xml). http://www.w3.org/XML.

[52] Wikipedia. ipad. http://en.wikipedia.org/wiki/Ipad.

[53] Wikipedia. ipad 2. http://en.wikipedia.org/wiki/IPad_2.

[54] Wikipedia. ipad (3rd generation). http://en.wikipedia.org/wiki/IPad_(3rd_generation).

[55] Wikipedia. ipad (4rd generation). http://en.wikipedia.org/wiki/IPad_(4th_generation).

[56] Wikipedia. ipad mini. http://en.wikipedia.org/wiki/IPad_Mini.

[57] Wikipedia. iphone. http://en.wikipedia.org/wiki/IPhone_(1st_generation).

[58] Wikipedia. iphone 3gs. http://en.wikipedia.org/wiki/IPhone_3GS.

[59] Wikipedia. iphone 4. http://en.wikipedia.org/wiki/IPhone_4.

[60] Wikipedia. iphone 4s. http://en.wikipedia.org/wiki/IPhone_4S.

[61] Wikipedia. iphone 5. http://en.wikipedia.org/wiki/IPhone_5.

[62] Wikipedia. ipod touch. http://en.wikipedia.org/wiki/IPod_touch.

[63] Michael Wittman, Bernhard Efler, Tobias Dönz, and Martin Planer. Mozartspaces tutorial. http://www.mozartspaces.org/2.2-SNAPSHOT/docs/MozartSpaces_Tutorial.pdf.

[64] Xamarin. Mono. http://www.mono-project.com.

[65] Xamarin. Monotouch. http://xamarin.com/monotouch.

All web references have been last accessed on August 11, 2013.