FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Energy-efficient Persistence for Extensible Virtual Shared Memory on the Android Operating System

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jan Zarnikov
Matrikelnummer 0426379

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, 30. April 2012

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Energy-efficient Persistence for Extensible Virtual Shared Memory on the Android Operating System

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Jan Zarnikov
Registration Number 0426379

to the Faculty of Informatics
at the Vienna University of Technology

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

Vienna, 30. April 2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Jan Zarnikov
Baumgasse 29-31/66/9, 1030 Wien

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

(Ort, Datum) (Unterschrift Verfasserin)

Danksagung

Diese Diplomarbeit und mein Informatikstudium an der Technischen Universität Wien wären ohne Unterstützung durch gewisse Personen nicht möglich gewesen.

In erster Linie möchte ich mich bei meinen Eltern bedanken, die mich mit viel Geduld während des gesamten Studiums unterstützt haben. Ich möchte mich auch bei allen bedanken, die mein Auslandssemester in Schweden möglich gemacht haben. Es war ohne Zweifel das Highlight meines Studiums.

Und nicht zuletzt möchte ich mich auch bei eva Kühn und Tobias Dönz bedanken, die mich beim Schreiben dieser Arbeit betreut haben.

Abstract

The result of the rapid growth of the Internet is that most computers in developed countries are always online. This always-connected state is now also spreading to mobile devices. This allows us to develop a new generation of distributed and mobile applications. However coordination of processes within a distributed application is a difficult task. Space based computing solves the coordination problem by writing information into a virtual shared medium called the space. The space (sometimes also called blackboard) decouples the processes and eliminates explicit communication between them.

eXtensible Virtual Shared Memory (XVSM) is a specification of a space based middleware. Until now MozartSpaces (the reference implementation of XVSM) did not support persistence which means that all data was lost when the space was shut down or crashed. Previously it was attempted to solve this problem by either implementing persistence as orthogonal functionality with aspects or by replacing the core of MozartSpaces with a database. Unfortunately both approaches have serious drawbacks: the incompatibility with the XVSM transactions and in the case of the aspect-base solution the problematic management of entry IDs. Therefore a new approach has to be found. Also to our knowledge there is currently no space based middleware that can run and store data on a mobile platform without the need for a server.

The solution proposed in this thesis is a new persistence layer which is tightly integrated into the core of MozartSpaces. The persistence layer hides the database-specific details and can use different database engines. Currently supported databases are Berkeley DB and SQLite but it is possible to add support for further databases.

A benchmark suite was developed to evaluate the performance of the persistence. The results show that both Berkeley DB and SQLite are very fast. A nice side effect of using a database to store data is a significantly reduced memory footprint at runtime. An important requirement was also the energy-efficiency of the persistence on mobile devices running the Android operating system. This goal was reached only partially because there are no energy profilers that can measure the energy consumption of all relevant components on a scale small enough to be usable for code optimization.

Overall the new persistence implementation works very well. The new extended version of MozartSpaces offers good performance and remains compliant with the XVSM specification.

Kurzfassung

Das Ergebnis des rapiden Wachstums des Internets ist, dass die meisten Computer in entwickelten Ländern immer online sind. Dieser Zustand des Immer-verbunden-seins breitet sich jetzt auch auf mobile Geräte aus. Das ermöglicht uns eine neue Generation von verteilten mobilen Anwendungen zu entwickeln. Die Koordination der Prozesse einer verteilten Anwendung ist eine schwierige Aufgabe. Space Based Computing löst das Problem der Koordination indem Informationen in ein gemeinsames virtuelles Medium namens Space geschrieben werden. Der Space (manchmal auch Blackboard genannt) entkoppelt die Prozesse und eliminiert explizite Kommunikation zwischen ihnen.

eXtensible Virtual Shared Memory (XVSM) ist eine Spezifikation einer Space-Based-Middleware. Bis jetzt hat MozartSpaces (eine Implementierung von XVSM) keine Persistenz unterstützt, was bedeutet, dass alle Daten verloren gingen, wenn der Space abgeschaltet wurde oder abgestürzt ist. Es wurde bereits versucht dieses Problem zu lösen, indem die Persistenz als orthogonale Funktionalität mit Aspekten implementiert wurde oder indem der Kern von MozartSpaces durch eine Datenbank ersetzt wurde. Es hat sich leider gezeigt dass beide Ansätze Nachteile mit sich bringen, nämlich die Inkompatibilität zu den XVSM-Transaktionen und im Falle der aspektbasierten Lösung das problematische Management der IDs der Einträge. Weiters gibt es derzeit nach unserer Kenntnis keine Space-Based-Middleware, die auf mobilen Geräten laufen und Daten speichern könnte ohne dabei einen Server zu benötigen.

Die in dieser Diplomarbeit vorgeschlagene Lösung ist eine neue Persistenzschicht, die in den Kern von MozartSpaces integriert ist. Die Persistenzschicht versteckt die datenbank-spezifischen Details und kann unterschiedliche Datenbanken verwenden. Derzeit werden Berkeley DB und SQLite unterstützt aber die Unterstützung weiterer Datenbanken kann einfach hinzugefügt werden.

Eine Reihe von Benchmarks wurde entwickelt um die Leistung der Persistenz zu evaluieren. Die Ergebnisse zeigen, dass sowohl Berkeley DB als auch SQLite sehr schnell sind. Eine schöner Nebeneffekt bei der Verwendung einer Datenbank ist, dass der Speicherverbrauch zur Laufzeit gesenkt werden konnte. Eine wichtige Anforderung war auch, dass die Persistenz auf mobilen Geräten mit dem Android-Betriebssystem stromsparend ist. Dieses Ziel wurde nur teilweise erreicht, da die verfügbaren Energie-

Profiler nicht den Stromverbrauch aller relevanten Komponenten messen können. Weiters ist die Granularität der Messungen für Code-Optimierungen nicht fein genug.

Insgesamt funktioniert die neue Implementierung der Persistenz sehr gut. Die neue erweiterte Version von MozartSpaces bietet eine gut Performance ohne dabei die XVSM-Spezifikation zu verletzen.

Contents

List of Figures

List of Tables

List of Abbreviations

ACID	Atomicity, Consistency, Isolation and Durability
CAPI	Core API
FIFO	First In, First Out
HDD	Hard Disk Drive
J2SE	Java 2 Standard Edition
JVM	Java Virtual Machine
LIFO	Last In, First Out
NoSQL	Not Only SQL
ORM	Object Relational Mapping
P2P	Peer-to-peer
POJO	Plain Old Java Object
SBC	Space based computing
SQL	Structured Query Language
SSD	Solid State Drive
SXQ	Simple XVSM Query
XQ	XVSM Query
XVSM	eXtensible Virtual Shared Memory
XVSMP	XVSM Protocol

CHAPTER 1

Introduction

The internet has become ubiquitous medium. It connects not only desktop computers and servers but also more and more mobile devices. Smartphones are becoming the norm and even the internet-enabled refrigerator, once a futuristic anecdote, is now reality.

This new generation of devices is accompanied by new programming paradigms. Current distributed applications are based on the client-server model. The communication between the server and the clients is usually realized with remote method calls or some form of asynchronous message passing. The drawback of this approach is the scalability of the server. Increasing number of clients can easily bring the server to its limits. Another problem is the reliability of such solution. If the server becomes unavailable (e.g. because of a hardware failure) the whole distributed application stops working. Furthermore the client is degraded to a simple consumer of the work done by the server although the hardware is often capable of so much more.

Peer-to-peer (P2P) architectures divide the problem at hand to several tasks which are processed by several more or less equally important nodes. In an ideal P2P architecture there is no central server and therefore no single point of failure. This approach leads to a more effective use of resources because the work can be split among the nodes. There is no busy server and idly waiting clients.

The drawback if P2P is that the communication and coordination gets very complex. Without any central coordinator the decision which node should do what becomes difficult. The communication and sharing of data between the nodes is also complicated.

1.1 Space Based Computing

Space based computing is an interesting approach that contributes to the problem of coordination and communication in distributed applications. It can be seen as an alternative to the classical client-server approach and P2P architecture. It was introduced by David Gelernter in [Gel85] as the **Linda** programming language. The principle is quite simple yet very powerful. The data in form of tuples (a vector of fields) is stored in a so called tuple space. Several processes (local or remote) can access the space at the same time. Tuples reside in the space and are not bound to any particular process of the distributed application. Space based computing decouples the data from the process. Gelernter defined three basic operations on a space:

- Out - writes a new tuple into the space, sometimes also referred to as the "insert" operation.
- In - searches for a tuple that matches the search criteria which are specified using a template. A template is a sample tuple where the fields have the same values as the searched tuple or null if the field and its value are not relevant for the search. If a matching tuple is found then it is returned and removed from the space. Otherwise the process blocks until a matching tuple is available. If the template matches several tuples in the space then one of them is returned indeterministically. This operation is sometimes also called "take".
- Rd - similar to the take operation except the tuple is not removed from the space, sometimes also called "read".

All three operations are atomic. This is especially important in the case of the take operation (sometimes also called consuming read). Also note that there is no update operation. Tuples are immutable and once they are written into the space they cannot be changed.

The space based computing paradigm and how it can be applied to complex distributed applications is described in more detail in [Mor10]. Several implementations of the tuple space concept exist coming both from academia and commercial software vendors. A good overview of the different implementations can be found in [WCC04].

eXtensible Virtual Shared Memory (XVSM) takes the concept of Linda as described in [Gel85] and other tuple space based systems and extends it by adding some new features.

1.2 Motivation and goals

Hardware of mobile and handheld devices has made great progress in recent years. Modern mobile phones have the same computing power like normal desktop computers

had just few years ago. The speed of various wireless technologies they use to connect to the internet is constantly increasing. This allows us to run complex and often distributed applications on a device that can fit into you pocket. Of course development of distributed mobile applications is not really possible without proper tool support. A space based middleware that can run on both mobile devices and "normal" computers is a tool that significantly reduces the complexity of distributed applications.

XVSM (and space based middlewares in general) is not intended to be a database replacement. It is not designed to hold and manage large quantities of data. It is not supposed to evaluate complex queries. The main goal of space based computing is coordination of distributed processes by sharing data.

Currently the reference implementation of XVSM does not support persistent data storage. Previous attempts to solve this problem by implementing persistence as orthogonal functionality with aspects [Mei11] and by replacing parts of the middleware with a database [Bar10] have shown that these two approaches have some serious drawbacks. The goal of this thesis is to find a new solution for this problem. The main requirements are:

- Extend the reference implementation of XVSM by providing a new persistence layer that can be used to store the state of the space.
- It must be possible to completely restore the last consistent state of the space after the space was shut down or crashed.
- The persistence layer should be based on a well tested and well documented database engine.
- The persistence must not be tied to one particular database engine. It must be flexible and extensible. It should be possible to use different databases depending on the application scenario and it should be easy to add support for new databases if needed.
- It is important that the persistence is compatible with both the Java and the Android version of the XVSM implementation.
- The persistence implementation should be fast. A set of performance benchmarks has to be developed to evaluate the speed of XVSM with persistence and compare it to previous versions.
- When run on a battery powered mobile device the persistence must not consume too much energy. The energy efficiency must be evaluated using suitable tools and benchmarks.

1.3 Thesis Structure

This thesis is organized as follows: Chapter 2 describes XVSM, its formal specification and the current status of the implementation. It presents the basics of data persistence and energy efficient computing in general. Chapter 3 covers related work on the field of persistence in space based computing. Chapter 4 discusses the requirements on the data persistence for the Android and the Java 2 Standard Edition (J2SE) version of XVSM while Chapter 5 explains how these requirements have been realized as well as the technical details of the implementation. Evaluation of the performance and energy efficiency of the solution can be found in Chapter 6. Chapter 7 lists several open issues with persistence in XVSM. Finally, Chapter 8 summarizes and closes this thesis.

CHAPTER 2

Background

This chapter covers several topics that are important for the understanding of the problem statement and the solution presented in this thesis. First the space based middleware XVSM is described followed by a brief introduction into persistent data storage. Further this chapter covers the basics of energy efficient computing on mobile devices and data storage on mobile devices, especially devices running the Android operating system.

2.1 XVSM

Description of XVSM

eXtensible Virtual Shared Memory (XVSM) is developed by the Space Based Computing Group of the Institute of Computer Languages at the Vienna University of Technology.

Formal specification

The complete formal specification of XVSM can be found in [KMS08, CKS09, Cra10]. Here is a brief overview.

The general architecture of XVSM is shown in Figure 2.1. The XVSM algebra defines the basic data-structures and operations on them. The functionality is divided into four layers CAPI-1 to CAPI-4 where each layer uses only the services of the layer bellow. Applications access the space using a language independent protocol (XVSMP).

XVSM algebra XVSM operates on a hierarchical data-structure called xtree that contains collections and unstructured objects such as numbers and strings. There are two

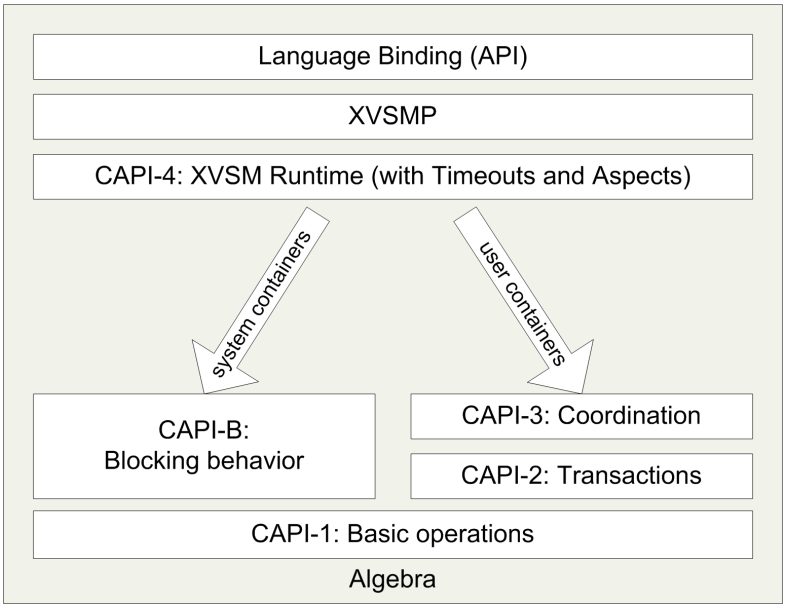


Figure 2.1: Architectural overview of XVSM [Cra10]

types of collections: ordered lists and unordered multisets. A collection can contain unstructured objects such as numbers or strings or another collection. Each object or collection has an optional label if it needs to be accessed directly. Substructures of an xtree can be accessed using the labels (or alternatively indices in case of ordered lists) of the path from the root of the xtree to the desired substructure. The labels do not have to be distinct, however using identical labels for several elements within a collection leads to indeterministic behavior.

An *entry* consists of one or more *properties* which are label-value pairs. The value of a property can be either an unstructured object or an xtree. A *container* is a multisets xtree that holds entries. An XVSM *space* is an xtree that contains all local containers with each container having a unique label.

XVSM Query Language XVSMQL allows more advanced selection of substructures of an xtree than just by labels or indices. An *XVSM Query* (XQ) consists of one or more *Simple XVSM Queries* (SXQ) that are combined using the pipe operator. It is a filter chain where the results of a SXQ are used as input for the next SXQ. A SXQ can filter or reorder the data before passing it to the next SXQ in the chain. SXQ cannot introduce new data.

XVSM offers several predefined SXQ, some of which can be parameterized:

- *cnt(n)* for selecting n entries.

- Sorting the entries according to a specific property value.
- Reversing the ordering of the entries.
- Eliminating duplicate entries based on a specific property value.
- Selecting one/all entries having a property value within/outside a certain range expression.

CAPI-1: Basic operations The lowest CAPI (Core API) layer offers simple storage facilities. It provides a write operation as well as read and take operations using a XVSMQL query. This storage is used both for the user entries and the metadata required by the runtime itself.

When writing an entry the label can be either explicitly defined or automatically assigned in which case it is returned as the result of the write operation. CAPI-1 also defines a bulk write operation for convenience reasons. If the query of a read or take operation does not match anything a special status code is returned indicating that the operation should be retried later.

CAPI-2: Transactions The second layer adds transaction support for executing several CAPI-1 operations as a single atomic action. XVSM uses pessimistic locking to achieve ACID properties (atomicity, consistency, isolation and durability) as defined in [Gra81]. A transaction can span across several requests and the processing of each request is encapsulated in a separate sub-transaction.

CAPI-3: Coordination Operations of CAPI-3 are synchronous, non-blocking and always return one of the following return codes:

- `OK` indicates that the operation was executed successfully.
- `DELAYABLE` means that the operation cannot be executed right now and should be retried later.
- `LOCKED` is returned when the resources needed to finish this operation are locked by another transaction.
- `NOTOK` is returned when the execution of the operation failed.

The third layer adds features that distinguish XVSM from a simple tuple space, namely the concept of coordinators. A coordinator decides how entries are stored and retrieved from a container. Different applications require different data management styles. XVSM allows user defined coordinators but many use cases can be covered with the predefined coordinators:

- **QueryCoordinator** which can be used to issue direct queries in XVSMQL.
- **FiFoCoordinator** makes sure that queries are read in the exact same order in which they were written by attaching a sequence number to each entry.
- **LiFoCoordiantor** is similar to the FiFoCoordinator but uses first-in-last-out ordering.
- **KeyCoordinator** turns the container into a key-value store. The keys (not to be confused with labels) have to be unique. Writing an entry with a key that is already present in the container or reading an entry with a key that is not present will cause a `DELAYABLE` result status.
- **LabelCoordinator** is similar to the KeyCoordinator except that labels unlike keys do not have to be unique.
- **LindaCoordinator** implements the template matching of entries described in [Gel85]. Additionally it supports retrieving more entries that match the given template.
- **VectorCoordinator** organizes the entries as an ordered list. New entries can be appended to the list or inserted at a specified position shifting all subsequent entries to the right. Entries can be selected using their index. Removing an entry causes all subsequent entries to shift to the left in order to avoid an empty gap.

A coordinator can store its own metadata for internal bookkeeping in a special xtree separate from user data or it can attach the metadata to the entries. A coordinator can define its own constraints on the entries it handles.

CAPI-3 also provides methods for creating and destroying containers. When a new container is created a list of coordinators and an optional size limit are specified. Additionally each container has a SystemCoordinator that is responsible for checking that the number of entries does not exceed the size limit of the container.

CAPI-B If a CAPI-3 operation cannot be executed immediately (e.g. a take operation on an empty container) it returns with an appropriate status code. CAPI-B provides an alternative behavior - operations can block until they can be carried out. CAPI-B is only used for metadata of XVSM and it is not visible to the user.

CAPI-4: Runtime with timeouts and aspects CAPI-4 defines the XVSM runtime which can accept local or remote operation requests. CAPI-4 is responsible for scheduling incoming requests and generating the responses.

CAPI-3 operations are non-blocking and always return immediately. CAPI-4 adds blocking behavior with timeouts. A response is sent immediately if the result of a CAPI-3 operation is `OK` or `NOTOK`. If the result is `DELAYABLE` or `LOCKED` then the request is queued. There are two ways how the request can be removed from the queue. Either the state of the space changes (e.g. the conflicting lock is released) and the request can be processed or the specified timeout is reached.

The runtime also adds aspects which provide a way how to extend the functionality. XVSM offers several interception points before and after the execution of the following space operations where user-defined code can be inserted:

- Writing, reading, deleting and taking of entries from a container.
- Creating, looking up, locking and deleting of a container.
- Start, commit and rollback of a transaction.
- Adding and removing of aspects.
- Shutdown of the space.

Aspects can be used to implement vertical functionality such as custom logging or security [KLM+97].

XVSMP XVSMP is a protocol that allows invocation of the CAPI-3 methods on a remote space. It is designed to be language independent. XVSMP uses asynchronous messages. Each request contains information about a response container where the result will be stored once the request has been processed.

Language bindings The language bindings make it possible to create XVSMP requests and read the responses. This is done by transforming calls from a language specific API to the language independent XVSMP.

Current status of implementation

The formal specification of XVSM is freely available and can be implemented by anyone without any licensing costs. It is language neutral and not bound to any specific hardware platform.

Haskell prototype

To evaluate the formal specification a prototype was implemented in Haskell [Cra10]. It is intended as reference for future implementations by defining exact semantics. It is generally slower than other implementations in imperative languages.

MozartSpaces

The reference implementation of XVSM for Java Standard Edition (J2SE) called MozartSpaces is available under GNU Affero General Public License (AGPL) Version 3 at [15]. The development started in 2006 later followed by version 2 (currently active) which is a complete rewrite solving many functional and non-functional issues of the first version. Technical details of MozartSpaces version 2 can be found in [Dö11] and [Bar10].

Unlike most tuple space implementations MozartSpaces allows objects of arbitrary classes to be stored as tuples as long as they are serializable. This allows much cleaner application design because it eliminates the need for conversion between tuples and objects from the domain model.

Android

The code of MozartSpaces is the basis for the Android implementation of XVSM [Luk]. Most of the code base remains the same since Android applications are also written in Java. Because of limitations of the Android OS (see Section 4.4) some changes have to be made while retaining full compatibility with the J2SE version.

.Net

An alternative implementation in .Net is currently being developed [19]. The goal is to have a complete implementation of the specification and not just language bindings. The result should be fully compatible and interoperable with MozartSpaces.

TinySpaces - .Net Micro Framework

TinySpaces is an alternative Implementation of the XVSM Specification. It uses the .Net Micro Framework is targeted at embedded devices. Unlike other mobile space based middlewares such as MobiSpace [FT05] or MobileSpaces [RD05] TinySpaces does not require a server. The space runs completely on the embedded device. In order to achieve reasonable performance some parts of the specification had to be omitted or simplified [Mar10].

iOS and other platforms

Devices running the iOS are currently very popular. Having an interoperable implementation of the XVSM specification that runs on this platform would be desirable. There is already an (incomplete) prototype implementation of the CAPI-3 interface.

There are many other popular software platforms but with current resources of Space Based Computing Group and volunteering contributors it is not feasible to provide an XVSM implementation for each platform.

2.2 Persistence

For performance reasons the state and data of a running application is held in RAM during execution. The content of RAM is lost when its power supply is interrupted or it is destroyed by the operating system when the application execution has stopped. Therefore applications have to make sure that their state and data are stored on some form of persistent medium.

Databases

Developing your own system for storing structured data from scratch is neither feasible nor necessary unless there are some really unique requirements. Most applications make use of an existing database component. A database is a system that allows storing and retrieving structured data. Depending on the type of the database there are constraints on how the data is structured and which storage and retrieval functions are available.

Besides simple storage some databases offer additional features such as complex data queries, transactions which guarantee that a set of operations will be executed as one atomic action or replication across several machines to increase performance and reliability.

Relational databases

The probably most common form of databases are relational databases. The concept goes back to 1970 when it was introduced by Codd in [Cod70]. Relational databases store data in the form of rows and columns in one or more tables. The definition of the tables and their columns is called schema of the database and defines the domain model of the data. Each table represents one entity of the model and each row in the database table is an instance of the entity. The columns and their types represent the attributes of the entity. The values in the table cells are either of primitive type or a reference to another row which is called relation (thus the name relational database). The schema helps to ensure data integrity.

The data manipulation is done through four basic operations on the data rows which are often referred to as CRUD: create, read, update and delete.

Relational databases can be queried using expressions in relational algebra. Most relational databases use the standardized Structured Query Language (SQL) for queries.

SQL is based on relational algebra but adds aggregation, grouping and arithmetic operations to extend its expressive power [Lib03].

NoSQL

An interesting alternative to relational databases which has become very popular in recent years are so called NoSQL databases. NoSQL stands for Not Only SQL.

In NoSQL databases the data does not have a predefined schema. The level of consistency and transactional security varies greatly. While some implementations offer ACID (Atomicity, Consistency, Isolation and Durability) properties others use the concept of eventual consistency. There are several basic types of NoSQL databases [Cat11]:

- **Key-value stores** which store a structured or unstructured value which can be retrieved using a key. Key-value stores are usually implemented as distributed hash-tables for better performance and scalability.
- **Document stores** which are similar to key-value stores but can persist more complex documents and support secondary indexes.
- **Extensible record stores** use rows and columns as datamodel. For better scalability the data can be partitioned not only horizontally but also vertically and distributed across several nodes.

The biggest advantage of the NoSQL approach is performance. Unlike relational databases which have to deal with all possible data models and different usage pattern the NoSQL databases are tailored for one particular application type. Because of this NoSQL can outperform classical relational databases in certain scenarios by a factor two [SMA+07]. On the other hand some sacrifices have to be made to achieve this performance. Depending on the particular NoSQL database the requirements on ACID properties, query language expressiveness or implicit data integrity using a schema have to be relaxed.

Persistence patterns

There is a mismatch between how data is represented in memory during application execution and the form in which it is stored in a relational database. Most applications are developed using object oriented programming. At runtime entities are represented as objects which consist of member variables of primitive datatype (integers, floating point number, strings etc.) or references to other objects. These objects, their member variables and references to other objects need to be converted to a form compatible with the database.

Object serialization

Serialization writes the state of an object to a data stream so that it can be stored in the file system or sent over the network to a different process. This means storing the values of all member variables of the object. When the object should be deserialized the process is simply reversed. The serialized object state can have a custom (usually binary) format or a more universal data formal such as XML. Modern software development environments such as Java or .Net provide object serialization as part of their standard libraries. One disadvantage of object serialization is that depending on the format stored objects cannot be queried as easily as for example relational databases.

Serialization of objects that consist only of primitive datatypes such as numbers or strings are quite simple. References to other objects make serialization more complex. There are several possibilities how to handle object references during serialization:

- Serialize the whole object graph. The result can be very large and possibly contain objects that are not required.
- Ignore object references and only serialize variables of primitive type. In this case the relationships between objects are lost upon deserialization.
- Let the developer choose which references should be serialized and restored upon deserialization.

Object relational mapping

Object Relational Mapping (ORM) allows developers to access entities stored in a database as if they were normal objects [O'N08]. The conversion between runtime objects and database representation is automatic and significantly reduces the need to write database-specific code. The most commonly used ORM frameworks map objects to SQL tables. Each member variable of an object is stored in one table column and references to other objects are mapped to foreign keys in the database. The ORM framework can also handle semantic incompatibilities, for example object oriented languages allow inheritance while SQL does not. Modern development environments such as Java and .Net provide feature-rich ORM frameworks.

2.3 Mobile energy-efficient computing

Energy-efficient computing has recently become a very popular research topic. There are two major reasons why energy-efficiency is important. For desktop computers and servers low energy consumption means saving money and simultaneously reducing the carbon footprint. On mobile devices however the motivation is increasing battery life.

One way how we can improve battery life is by improving the hardware. On the one hand battery capacity increases while the power consumption of the components is reduced. In case of mobile phones these advances have been mostly consumed by improving features. This means that new models are able to do more than the previous generation while the battery life remains more or less the same [Pen10].

The software architecture and programming style can also have an effect on the power consumption. Not only the operating system and device drive can improve energy efficiency but also user space application. In [Gar07] Garret argues that information polling in short cycles significantly increases the power consumption of the CPU and should be replaces by interrupt driven approach.

Another approach how to reduce power consumption presented in [KL10] is to offload complex computations to the cloud. Unfortunately this solution has two major drawbacks. First it creates communication overhead because all data necessary for the computation and sometimes even the computation algorithm itself as well as the result need to be transported over the network. Depending on the location and available wireless networks the bandwidth might be limited which increases time required for the computation. The other disadvantage is that sending and receiving data over wireless network consumes a lot of energy, often more than a mobile CPU [CH10].

2.4 Data storage on mobile devices

Mobile devices need to persistently store data just like any other computer. There are two basic ways how this can be accomplished. Either data is stored locally on a persistent medium such as a hard drive or a flash-based memory or it is stored remotely on a server and accessed through the network. Hybrid solutions that store data on a server and cache parts of the data locally are also an option [SLLP+10].

Local storage has the advantage that it is always available. Remotely stored data might become inaccessible if there is no network connection. The performance of remote storage depends on the performance of the network connection which can be a problem for mobile devices that often use slow and unreliable wireless networks. Modern wireless network technologies might offer sufficient throughput but the latency is still too high for effectively using remote resources [SBCD09]. Furthermore fast wireless networks are only available in densely populated urban areas.

On the other hand remote storage can offer unlimited capacity because it is not constrained by the size of the device. It is also cheaper (in terms of price per GB) because it does not have to be small and energy-efficient. Remote storage has also the advantage that it cannot be physically stolen as it is unfortunately sometimes the case with mobile phones.

There are many commercial providers of remote storage for mobile devices with Dropbox [9] and Amazon S3 [1] being probably the most popular ones.

Unlike desktop computers and servers which usually use spinning hard disk drives (HDD) most mobile devices rely on solid state flash based memory for local storage. There are several reasons for this. First of all the mechanical nature of HDDs and their size are not well suited for devices that are often moved. Depending on the type solid state drives (SSD) can offer higher throughput and lower latency compared to HDDs [RC10]. Flash memory also consumes significantly less power both under load and while idle [SHH10]. The only reason why many desktop computers and servers computers still use classic HDDs is much lower price per GB [LCKW].

The most common technology used for embedded devices is NAND flash memory. The main disadvantage of this memory type is that it does not allow bit-wise random write access. Data can be only written in block mode. Furthermore the number of times each block can be rewritten is limited. This means that access strategies typically used for HDDs have to be modified. These modifications need to be done on several levels – from the hardware controller to adapted databases and applications that are using them [STG10].

Local storage in Android OS

Most Android powered devices offer two local storage possibilities. Either applications can persist their data in the internal flash-based memory or they can access a removable external storage (usually a removable SD memory card). The internal storage uses either the YAFFS2 or the ext4 file system depending on the device model and Android version [2]. The external storage is usually formatted with the FAT32 file system for compatibility reasons.

Local storage on mobile phones running the Android OS was extensively analyzed in [KAU12]. The findings in this paper can be summarized as follows:

- The performance of the storage hardware can have huge implications on the speed of the application execution even in scenarios where one would not expect it such as a web browser.
- In most benchmarks the internal storage is just as fast as the best SD cards or slightly faster.
- There are huge differences between the performance of the various brands and models of SD cards. These differences have a noticeable impact on the user experience. The advertised speed class of the card is only loosely correlated to the actual speed.
- Slow storage can negatively influence the power consumption.

The paper also presents several possibilities how user experience can be improved by speeding up storage system. The most obvious improvement would be using faster flash

memory. Changes to the software stack on all levels (OS, the file system, the SQLite database and applications) would also improve the performance of the storage [KAU12].

AndroBench [18] is a tool for measuring the performance of the storage of a particular Android device. It measures sequential and random file access as well as the performance of the built-in SQLite database and then uploads the results to a central server. In [KK12] the authors of AndroBench present the performance data collected from various devices collected using the tool. Their results show that there is no clear winner and the performance depends on the concrete memory chip model used in the device. The choice of file system can also heavily influence the performance. The log-based YAFFS2 has faster random access while ext4 is faster during sequential access.

The information about the used memory chip model and file system is generally hidden from the application developer behind the public API. This can make it difficult to optimize the performance of the application. In [NK07] a specialized database for devices with flash storage is proposed which dynamically adapts its storage access strategy at runtime to achieve optimal performance for the given memory type.

CHAPTER 3

Related work

3.1 Persistence in space based computing

In the original Linda concept the tuples do not have an explicit well defined structure like a scheme in SQL databases. This makes efficient and flexible implementation of persistent tuple space difficult. This chapter gives an overview how different tuple spaces middlewares solved this problem.

Persistence in XVSM

The current version of MozartSpaces does not offer any persistence. However two older implementations of the XVSM specification did provide persistence.

Persistence implementation using space aspects

Aspects are one possibility how to implement persistence in XVSM. This is a very non-intrusive way because existing code does not need to be modified. An implementation of persistence using aspects in the first version of MozartSpaces is described in [Mei11].

In [Mei11] Meindl defines four persistence models in XVSM but only the first model is implemented:

A) The contents of the entire space is persisted.

B) Only selected containers are persisted.

C) Only the effects of operations in selected transactions are persisted

D) Only the effects of selected operations are persisted.

The persistence implementation defines a set of aspects that intercept the request processing and stores the data. Table 3.1 gives an overview of the aspect interception points and how they are used to persist data. XVSM transactions are mapped to database transactions. The persistence works with HSQL DB [16] and DB4o but is designed to be extensible and support for other database engines can be added easily.

Interception point	**Description**
post transaction create	creates a new database connection that correlates to the just created XVSM transaction
post transaction commit	looks up and commits the corresponding database transaction
post transaction rollback	looks up and rolls back the corresponding database transaction
post write	persists the entry that has just been written to the space
post take	deletes the entry that has just been taken from the space
post destroy	deletes the entry that has just been destroyed in the space
post container create	persists information about the container that has just been created in the space
post container destroy	deletes information about the container that has just been destroyed in the space
pre core shutdown	flushes the database if the space is flushed too

Table 3.1: Overview of interception points used by XVSM persistence implementation using aspects [Mei11]

During the development of the persistence several shortcomings of the first version of MozartSpaces were discovered, most notably:

- Some required interception points were not available (e.g. pre-shutdown, post-launch, aspects for adding and removing aspects)
- Because of the way how IDs were assigned to objects in the space (containers and entries) the IDs could not be consistently restored.
- Committing the database transaction in post-transaction-commit aspect violates the atomicity requirement because the space can crash after the MozartSpaces transaction has been committed but before the aspect invocation is completed.

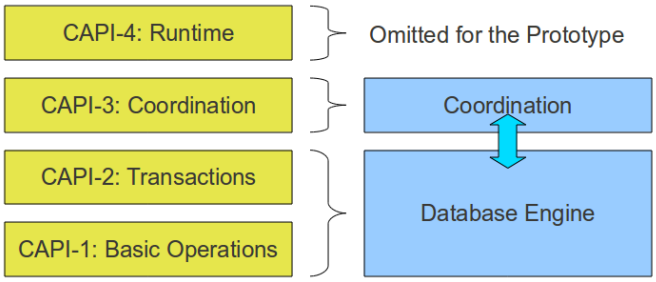


Figure 3.1: Architecture of XVSM implementation based on a database [Bar10]

XVSM implementation based on a database

An alternative way how to implement persistence in XVSM is to use a database engine as the lower CAPI layers [Bar10]. Figure 3.1 illustrates this approach. The basic idea is to reuse the functionality of the database because databases provide the same functionality as CAPI-1 and CAPI-2, namely simple storage facilities and transactions. Several database engines were evaluated and a prototype was build with the H2 Database engine [7].

Unfortunately the prototype has shown that this approach has some serious issues. First of all most databases have incompatible transaction semantics. XVSM ensures isolation by locking entries or if needed entire containers. The H2 Database can only lock whole tables. Furthermore it does not support subtransactions as described in the XVSM specification. In the end the prototype reimplemented huge parts of the transactional system instead of leveraging the existing functionality of the database.

Coordinators and XVSM queries posed another problem. The prototype stored coordination data (such as the labels of the LabelCoordinator or the ordering of FifoCoordinator) also in the database. Selecting an entry meant transforming the XVSM query into a SQL SELECT query and then running it against the database. This worked reasonably well for some coordinators and simple queries but more complex coordinators (such as LindaCoordinator) and XVSM queries with chained selectors led to extremely complex (and therefore slow) SQL queries which had to be combined with extra Java code [Bar10].

Conclusion of persistence implementations in XVSM

Experience has shown that neither aspects nor CAPI-3 built on a database are a good solution how to implement persistence in XVSM. Aspect-based persistence cannot be integrated very well into the space because aspects by design provide only loose coupling and hide the implementation of the space operations which is important for effective and efficient persistence implementation. A CAPI-3 built on top of a database is

also not a good solution because relational databases have different transaction semantics and XVSM queries cannot be converted into efficient SQL queries. Therefore a new approach has to be found.

Persistence in other space based middlewares

This section gives an overview of general purpose space based middlewares with goals and features comparable to MozartSpaces and how they implement data persistence. A more detailed comparison of different implementations can be found in [Mei11].

This overview does not contain GigaSpaces [33] because the primary focus of GigaSpaces is the development of scalable enterprise applications and the integration with various software frameworks. An analysis of this product and its persistence architecture would exceed the scope of this thesis.

Also not included in this overview are space based middlewares that run only on mobile devices such as MobiSpace [FT05] and MobileSpaces [RD05].

Persistent Linda

The original Linda programming language does not provide persistence or transactions. These were added as extensions in [AS92]. It also adds the possibility to update existing tuples in place. Otherwise the simplicity (and therefore its power to express parallel applications in an elegant way) remains the same.

JavaSpaces - Apache River

JavaSpaces was developed by Sun Microsystems (now Oracle Corporation) as part of the Jini project. Despite being technologically advanced Jini was not commercially successful. Therefore Sun decided to abandon it and release its source code under the open Apache Software License. Jini is now maintained by the Apache Software Foundation as Apache River [10]. A brief introduction into JavaSpaces is given in [20] and a complete documentation of JavaSpaces can be found in [FHA99].

In JavaSpaces tuples are represented by Java objects and their fields are automatically constructed from their public member variables. The operations read and take can specify for how long the process should block if no matching tuple was found using a timeout. JavaSpaces also allows to register notifications (a callback method) of future write operations of tuples matching a certain template.

Apache River can be run either in a transient mode where the content of the space is lost between executions or in a persistent mode where each space operation is serialized and written into a log in the file system. It is also possible to provide custom persistence implementation.

JavaSpaces - Blitz

Blitz is an alternative implementation of the JavaSpaces specification [6] which is available under a BSD license. Internally Blitz uses Berkeley DB to store data. There is no API for adding support for alternative database backends. Blitz can be configured to use on of the following three storage profiles:

- The **transient profile** uses the database to hold data at runtime but the contents is removed after restart which basically means that the space is not persistent. The reasons why it uses database nevertheless (instead of purely in-memory data structures) are reducing memory consumption at runtime by swapping the data between the memory and a database and code reuse with the next two profiles.
- The **persistent profile** persists the effects of all operations immediately.
- The **time barrier profile** guarantees that effects of space operations will be made persistent after a configured period of time. This can lead to data loss and inconsistent state after a crash. On the other hand it should provide better performance than the persistent profile.

The persistence profile is configured globally and it is not possible to use different profiles within one space. Blitz creates a new Berkeley DB container for each class used as entry type. Entries are transformed into a byte array using the standard Java serialization before they are written into the database. Blitz uses secondary indexes for faster template matching.

TSpaces

TSpaces were developed and later abandoned by IBM [LCX+01]. TSpaces is another tuple space implementation in Java. Compared with the original concept of Linda it offers several additional features. It supports timeouts and notifications similar to JavaSpaces. Furthermore the read and take operations can return all matched tuples and not just an indeterministically chosen one.

According to [28] TSpaces uses “a main memory database, and keeps it persistent with logging of DB writes/deletes”. Checkpoins of the data are made regularly to keep the log size small. IBM planned to add JDBC support to TSpaces. This feature was probably never implemented because the TSpaces project was abandoned by IBM.

Unfortunately TSpaces and its source code are no longer available on the project’s website and thus cannot be analyzed in more detail.

SQLSpaces

As the name suggests SQLSpaces [35] stores its data in a relational database. It consists of two parts - a server and a client. Both are written in Java but the client is also available for Prolog, C#, Ruby, PHP and Android operating system.

Tuples in SQLSpaces use a predefined class. They are basically a set of fields which must be of primitive type (integer, boolean, string etc.) or a binary blob or an XML document. Tuples are assigned an ID which can be used for updating the data stored in the fields. SQLSpaces support notifications on future write and delete operations and transactions which are based on transactions of the underlying database.

The default configuration uses an embedded HSQL database but also supports PostgreSQL and MySQL. SQLSpaces automatically creates a new database table for each new tuple signature (the types of the fields of the tuple). This is basically a form of ORM except that references to other objects are not allowed. The biggest advantage of this approach is that template matching can be implemented using an SQL query with a WHERE clause. On the other hand this is also the reason why entries must have the form of tuple of primitive fields.

TuCSoN

Tuple Centres Spread over the Network (TuCSoN) is a research project at the University of Bologna [21]. It is a coordination framework for distributed processes and mobile agents with focus on semantic data [NVP10].

As the name suggests an application using TuSCoN can be distributed across several nodes. TuSCoN offers some basic persistence which can be configured on a per-container basis. If enabled all operations of the given node are written into a logfile a custom serialization mechanism.

Other space based middlewares

An extensive complete analysis of persistence in various space based middlewares including additional products such as the commercial solution **GigaSpaces** can be found in [Mei11].

Summary

The persistence implementations of Apache River, Blitz SQLSpaces and TuCSoN are compared in Table 3.2. The table shows which persistence models (as defined in Section 3.1) are supported, whether different database engines are supported, how the data is converted into a persistable form and whether the middleware can store data on mobile devices.

TSpaces is not included in the table because the source code is no longer available. Persistent Linda was also omitted because there is currently no actively maintained implementation.

	Swapping	Persistence model				Different DBs	Data conversion	Mobile devices
		A	B	C	D			
SQLSpaces	Yes	Yes	No	No	No	Yes	Simple ORM	Only client
Apache River	No	Yes	No	No	No	No	Java serialization	No
Blitz	Yes	Yes	No	No	No	No	Java serialization	No
TuCSoN	No	Yes	Yes	No	No	No	Custom serialization	Only client

Table 3.2: Overview of persistence in different space based middlewares

As you can see the space based middlewares examined in this section use different approaches for data persistence. Only SQLSpaces support different databases and it is also the only examined space based middleware that uses object relational mapping rather than object serialization. While SQLSpaces offers an Android remote client none of the examined middlewares can actually run and store data on a mobile device.

CHAPTER 4

Requirements

This chapter summarizes both the functional and non-functional requirements on the persistence layer. The formal model of XVSM, its reference implementation MozartSpaces and the target platforms create many constraints that need to be considered during design and development of the persistence layer.

4.1 Requirements of the formal model of XVSM

XVSM is formally specified in [Cra10] and a brief overview of the specification was given in Section 2.1. This section focuses on the details of data persistence in XVSM.

The CAPI-1 layer provides a simple space for storing data. It offers three atomic operations: read, write and take. It does not support transactions. There are no further details on how this storage should be implemented. The persistence implementation must provide functionality to store individual entries.

The CAPI-2 layer provides transactions with ACID properties:

- Atomic – all operations within one transaction are executed as one action or not at all.
- Consistent – the data must be in consistent state when the transaction is committed.
- Isolated – concurrent not yet committed transactions must not interfere with each other. MozartSpaces supports the isolation levels read committed and repeatable read as defined in [ISO92].
- Durable – the effects of the operations are permanent.

Atomicity, consistency and isolation are achieved by pessimistic locking. Without persistent storage the durability requirement is not fulfilled. But atomicity and consistency are also important requirements for the persistence layer. Only data of committed transactions must be persisted. Changes made by yet uncommitted transactions or transactions that have been rolled back must not be stored. Persisting the changes of a committed transaction must happen as one atomic action, regardless of how many operations were encapsulated in the transaction. When the space is restored its state must be consistent. The implementation must ensure this even if the space crashes in the middle of a running transaction or in the process of committing a transaction.

The CAPI-3 layer adds coordination and container management. This means that not only individual entries need to be stored but also:

- **Container meta-data** – each container has a name, maximum capacity and a list of obligatory and optional coordinators. This information must also be persisted otherwise it would not be possible to restore the state of the space.
- **Coordination data** – most coordinators have some internal state that has to be persisted. This state can have different forms. Some coordinators store information on per-entry basis (e.g. KeyCoordinator, LabelCoordinator) while others have a global stage (e.g. the ordering of entries in VectorCoordinator).

To summarize the requirements of the formal model of XVSM the persistence layer must be flexible enough to store all kinds of data (entries, coordinator state and container meta-data) and it must be compatible with the transactional model of XVSM.

4.2 Persistence models

The four different persistence models have been described in [Mei11] and summarized in Section 3.1. The persistence implementation described in this thesis only needs to support the first two models. Unless stated otherwise the entire space should be persisted but it should be possible to declare selected containers as in-memory only.

The persistence models C (only effects of operations within selected transactions are persisted) and D (only effects of selected operations are persisted) have been omitted for two reasons. First it is difficult to find practical use cases for these models and second the behavior becomes difficult to understand for the user as illustrated by the following example. Imagine process A writes an entry using a persistent transaction and later process B deletes the entry using a non-persistent transaction. Should the entry now be visible to both processes or not? Should this even be possible? How can process B know that an entry written by another process is persistent or not? And what happens if the space is restarted? As you can see the persistence models C and D cannot be implemented until clear and intuitive semantics for such cases are defined.

4.3 Constraints of the MozartSpaces implementation of XVSM

MozartSpaces is currently considered as the reference implementation of the XVSM specification. For performance reasons MozartSpaces implements the layers CAPI-1 to CAPI-3 in a monolithic code structure. This means that MozartSpaces offers a XVSM-compliant CAPI-3 interface but internally there is no CAPI-2 or CAPI-1.

MozartSpaces is being actively developed by several contributors. To keep conflicts at a minimum the implementation of the persistence should change the architecture of MozartSpaces as little as possible. Of course some changes to the API are unavoidable. These changes must be discussed with the XVSM Technical Board [14] as they arise.

Support for large containers

Until now all data in the space was held in memory at all times using the Java Collections Framework. While this approach provides very good performance it does limit the scalability. Having all data in memory makes containers with many entries or spaces with many containers impossible. The persistence layer should move large portions of the data to persistent storage and only keep the necessary control structures in memory.

Platform requirements

MozartSpaces is written in pure Java and can run on any recent J2SE runtime. A slightly modified version of MozartSpaces was created for the Android operating system. Both versions are actively being developed and are interoperable. It is important that the persistence layer is compatible with both versions of MozartSpaces and offers reasonable performance on both platforms. Having some platform-specific code is unavoidable but it should be kept at a minimum.

The persistence of the J2SE version of MozartSpaces must remain pure Java or use a programming language that can be compiled into Java bytecode. Native libraries that require a specific hardware platform or operating system should be avoided.

Flexibility

There is no one-size-fits-all persistence solution. Different application scenarios have different requirements. The persistence layer must be flexible and configurable to suite different needs. Some applications require a high throughput, some applications use very large entries, some applications have many entries or containers and the size of transactions (the number of operations within a transaction) can vary as well. Clearly it is not possible to achieve excellent performance under all possible circumstances.

But it should be possible to configure different parts of the system to achieve optimal performance for a particular use case.

In-memory mode

Despite the introduction of persistence it should still be possible to run MozartSpaces in an in-memory mode. This mode should have the same behavior and performance as the last version of MozartSpaces without the persistence layer.

It should also be possible to create in-memory-only containers while other containers within the same space use persistent storage.

Modularity and future extensibility

MozartSpaces is divided into several modules some of which are optional (depending on application scenario). The architecture of the persistence should follow this approach. Functionality should be split into separate modules with clear boundaries and interfaces. It should be possible to extend the functionality (for example add support for data replication) with additional optional modules (plugins) without having to change the existing code.

Code quality and maintenance

MozartSpaces already employs both unit and integration tests. These can be used to ensure that the necessary code changes made will not change the behavior in an unexpected way. New tests must be developed to verify the functionality of the persistence layer.

MozartSpaces also uses Checkstyle [27] to ensure that the source code follows coding standards. These standards ensure that all developers of the project use the same style when writing code. This leads to better readability which makes maintenance easier. Checkstyle is also used to ensure that all classes and methods are properly documented in JavaDoc.

4.4 Constraints of the Android operating system

The Android operating system as target platform introduces additional constraints on the implementation. The Android OS is designed for embedded and handheld mobile devices with limited hardware resources. Compared with normal computers these devices often:

- have just one CPU core. Although dual-core Android phones started to show up on the market in the recent months they are still very expensive and rather an

exception. Furthermore ca. 67% of users use Android 2.2 or older [13] which does not support multiple cores.

Having only one core reduces performance gains of multithreading which in return has negative effects on scalability.

- have generally a much slower CPU. On mobile devices size, energy consumption and heat production of the CPU are much more important than on desktop computers. To satisfy these requirement manufactures build mobile devices with smaller, more efficient but generally slower CPUs based on the ARM-architecture.
- have less and slower operating memory. Just like with the CPU sacrifices have to be made with the operating memory in order to make devices truly mobile.
- have less memory for permanent storage.
- are connected using various wireless technologies such as Wi-Fi, Bluetooth or cellular network that offer limited speeds and poor reliability.
- are battery powered. On one hand consumers want mobile handheld devices to be as small and light as possible while on the other hand the device should be usable for as long as possible without recharging. These two requirements are clearly contradictory and manufactures have to make a compromise between battery life and battery size (and thus the size of the device). On a typical smartphone there are four components that are responsible for most of the power consumption:

  1. Wireless connectivity - most Android devices offer various wireless technologies such as Bluetooth, WI-FI or GSM/UMTS mobile network. Although these technologies offer different speeds, latency and reliability they have one thing in common - they consume energy when receiving and especially when sending data.
  2. Display - modern displays on smartphones and tablets offer high resolution and bright colors. Despite major improvements the display is still draining a lot of energy when turned on.
  3. Computing resources - heavy load on the CPU and RAM over longer periods of time can seriously drain the battery of most mobile devices. Fighting this kind of energy consumption is complicated because it is usually a “death by thousand cuts”. Often there is no single line of code causing the energy drain, it is the overall designing and programming style that affects the energy efficiency.
  4. Storage - usually there is some internal flash memory for persistent storage of data. Some devices also offer additional storage options such as removable memory cards or external USB drives.

> The significance of the four points described above varies across devices and depends on the usage patterns and applications that run on the device [CH10].

The limitations described above come from the hardware of the devices that Android OS was developed for. The operating system itself, the development tools and the API have further limitations:

Android development tools use Java as programming language. Because of licensing issues the source code is not compiled to Java Virual Machine (JVM) bytecode but to a different format called Dalvik executables. Dalvik is a virtual machine specially designed for devices with limited hardware resources. JVM bytecode cannot be run on the DalvikVM . Static conversion of JVM bytecode to Dalvik executables is possible. This means that tools and libraries that use bytecode generation and manipulation at runtime such as ASM or CGLib cannot be used on Android.

The Android API has some limitations. It does not offer the full class library of Java 2 Standard Edition. Instead a subset of Apache Harmony (an alternative implementation of the J2SE class library with open-source license) is used. Compared to the full class library of J2SE the following parts are missing: AWT and Swing GUI libraries, JAXB, RMI, JNDI, ImageIO, XA-Transactions, JavaBeans, MXBeans, Corba, Printing and parts of Security [3]. This can be an issue when porting existing Java applications to Android.

Writing native applications in C or C++ is also possible but it makes distribution and deployment of applications more complex because a separate build has to be created for each processor type (e.g. not all devices have a FPU).

CHAPTER 5

Implementation

This chapter describes how the persistence layer of MozartSpaces was implemented. The general architecture of MozartSpaces remains the same. The description of the implementation of the lower layers (CAPI-1 to CAPI-3 of the XVSM specification) can be found in [Bar10] and the upper layers (CAPI-4 and XVSMP) are described in [Dö11].

The persistence uses a local embedded database. The reason for this decision is the high latency and low reliability of network connections on mobile devices. Local storage offers much lower latency and is always available regardless of the network status. The latest generation of smartphones offers several GB of space which means that the storage capacity should not be an issue.

It was decided that the transactional isolation of MozartSpaces will remain unchanged. This means that isolation is accomplished by pessimistic locking of entries and containers as described in [Bar10]. Transactional isolation of the database (if it offers any) is not used. While this might look like a missed code-reuse oportunity previous experience has shown that reusing the isolation offered by the database does not yield the desired results [Bar10, Mei11].

In [Mei11] the persistence is implemented as orthogonal functionality using aspects. The advantage of this approach is loose coupling between the persistence and the core of MozartSpaces. Unfortunately it has some limitations because in some cases a tighter integration of the persistence in MozartSpaces is necessary. Because of this it was decided to implement the persistence inside the core of MozartSpaces and integrate it with the CAPI-1 to CAPI-3 layers. While this means a tight coupling between the persistence and the core of MozartSpaces it allows a clean implementation that does not violate the XVSM specification.

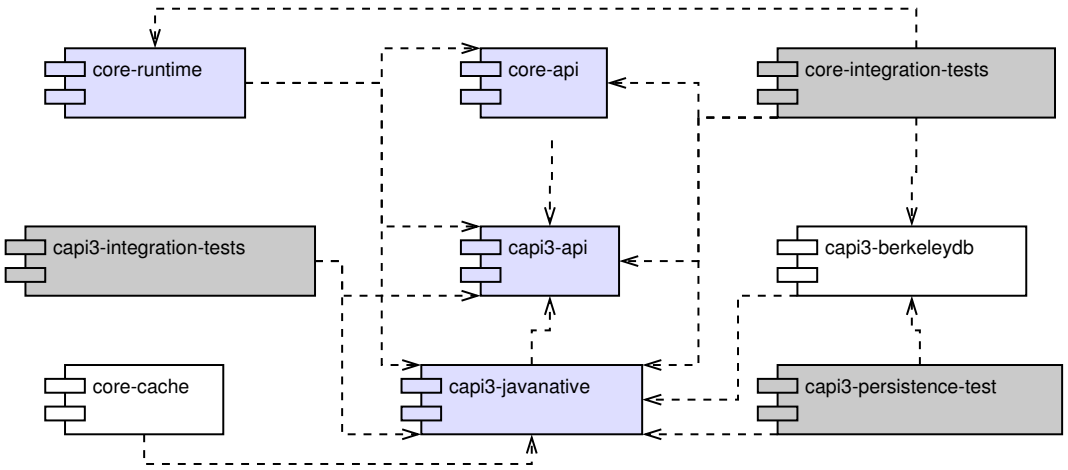


Figure 5.1: Components of MozartSpaces relevant to the persistence and dependencies between them. Blue components are always deployed and used at runtime, gray components contain only test cases and are never used at runtime and white components can be optionally used at runtime if their functionality is required.

5.1 Architecture overview

MozartSpaces is built using Maven and consists of several modules. Figure 5.1 shows the dependencies between modules. Some modules have been omitted from the diagram (such as the examples module, XVSMP, network communication module and a shared module with utility classes) because they are not relevant for the persistence.

core-api contains the public API of MozartSpaces core that can be used by client applications that use the space in embedded mode.

core-runtime contains the implementation of the CAPI-4 layer of the XVSM specification.

core-integration-tests contains integration tests for the whole MozartSpaces core.

capi3-api contains the public API of the CAPI-3 layer of the XVSM specification.

capi3-javanative contains the Javanative implementation of the capi3-api. This is the component where the most changes have been done by the persistence layer. It contains code of the persistence layer that does not depend on any particular persistence backend.

capi3-berkeleydb contains the code of the persistence backend based on Berkeley DB (see Section 5.2)

capi3-persistence-tests contains tests for the implementation of the persistence layer. These tests verify the functionality of the persistence in different configurations on a low level.

capi3-integration-tests contains integration tests for capi3-javanative implementation.

core-cache contains optional configurable caches for different parts of the system (see Section 5.5).

The class diagram in Figure 5.2 shows how the persistence is integrated with the rest of the Java native implementation. Figure 5.3 shows a class diagram focusing on the implementation of the persistence. The purpose and implementation of the most important classes will be explained throughout this chapter.

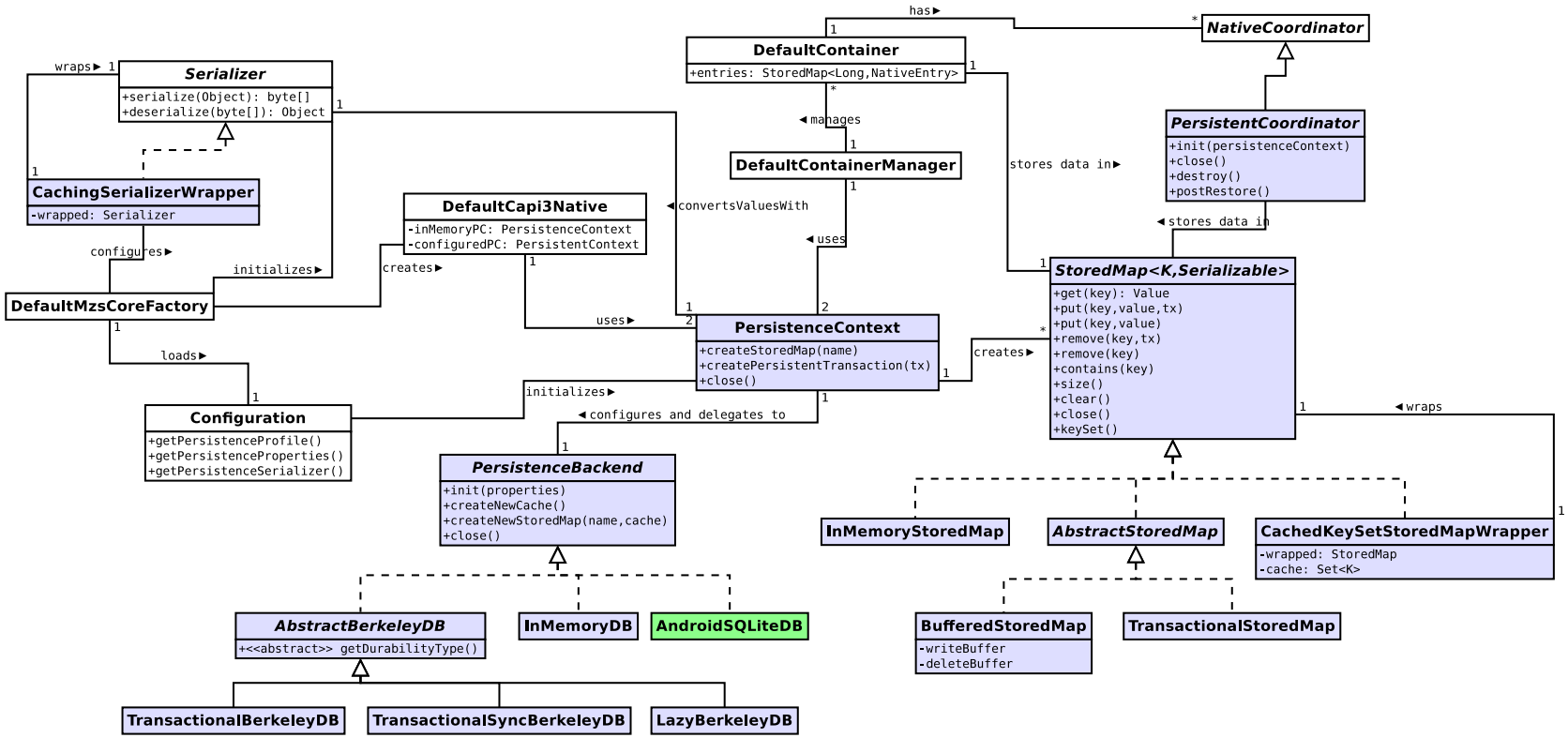


Figure 5.2: Class diagram with the most important classes and interfaces of the persistence layer. Classes that already existed in previous versions of MozartSpaces are white, new classes are blue and classes specific to the Android version are green.

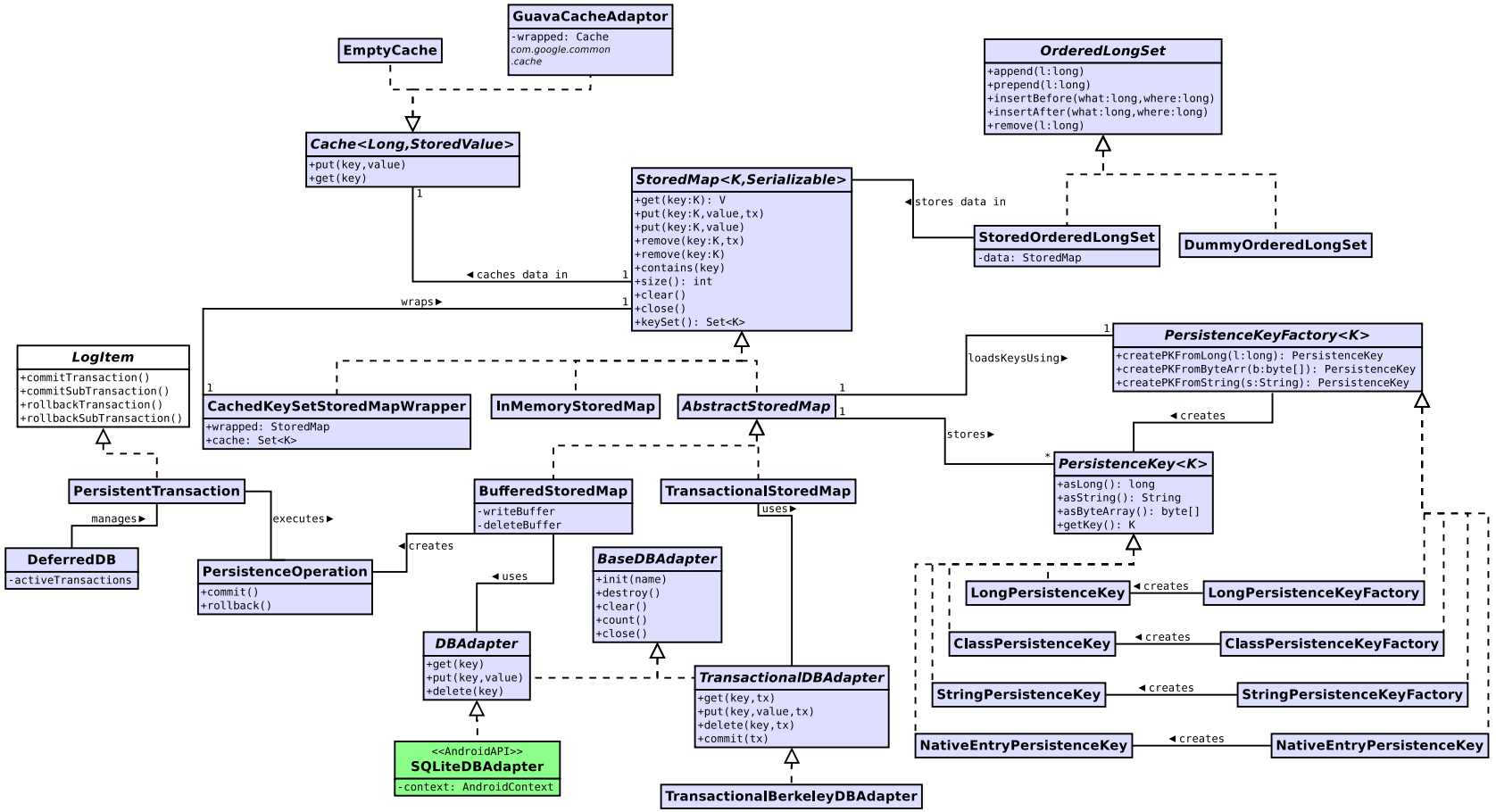


Figure 5.3: Class diagram showing the `StoredMap`, its implementations and related classes and interfaces. Classes that already existed in previous versions of MozartSpaces are white, new classes are blue and classes specific to the Android version are green.

5.2 Storage backend

The goal of the persistence layer is to replace the existing Java collections with something that a) offers persistent storage of data, b) does not keep all data in memory and c) offers a similar API in order to keep changes to existing code base at a minimum.

Choosing a storage engine

Developing a completely new persistence layer from scratch is neither feasible nor reasonable. It was decided to use an existing storage engine and integrate it into MozartSpaces. Several options were evaluated based on these requirements:

- The storage engine does not have to provide isolation of transactions because this is already handled by MozartSpaces. In fact it must be possible to read uncommitted data written by concurrent transactions. Most databases have their own semantics of transactional isolation which are not compatible with the specification of CAPI-3.
- The storage engine does not have to manage the conversion of Java objects to a persistent data format (such as SQL rows and columns) and vice versa. All objects managed by the space are already serializable because they have to transfered over the network. MozartSpaces offers several different serialization mechanisms that can be reused for the persistent storage.
- The storage engine does not have to handle relationships between objects. The entries and coordination data stored in the space do not have references to each other.
- The storage engine should provide atomicity. This means that it should be possible to group several write operations into a single atomic action.
- The storage engine must run on the J2SE platform and must not use any native code which would limit the usage of MozartSpaces to a specific hardware or operating system.
- The storage engine must run on the Android operating system.
- using the same storage engine in both the Java version and the Android version of MozartSpaces is desirable but is not really necessary.
- The storage engine should have a small memory footprint and it should not require too many transient dependencies (ideally none). This is especially important for the Android version in order to keep the application small.

- The license of the storage engine must be compatible with the license of MozartSpaces (AGPL) and it's distribution channels.

Several storage engines have been evaluated and two matched the criteria described above:

Berkeley DB Java Edition

Berkeley DB perfectly matches the requirements described above. It is written in pure Java and runs both on J2SE and Android OS. In fact there are two versions: one for J2SE and one for Android OS but they use the same API and the differences between those two are not visible to the application developer. The Android version of Berkeley DB stores the data in the external storage (such as SD memory card). Berkeley DB is a NoSQL database. It is basically a simple key-value store. Both keys and values are untyped and can be accessed as byte arrays. The currently latest version of Berkeley DB (5.0.34) was used during development and evaluation of the persistence layer.

Android SQLite

The Android OS comes with a built-in SQLite database support. It is part of the standard Android Java API and does not require any additional libraries. Depending on the version of Android OS different version of SQLite is included: 3.5.9 (Android 1.5+), 3.6.22 (Android 2.2+) or 3.7.4 (Android 3.0+). As the name suggests SQLite is a lightweight database. It implements most of the SQL92 standard. The parts of SQL92 that have been omitted [31] are not necessary for the implementation of the persistence. It is written in C and runs natively on the Android OS. SQLite stores its data in the device's internal memory and not in the external storage. This has two advantages: it does not require any permissions and the data is protected and cannot be accessed by other potentially malicious third party applications.

Alternatives

Few other storage engines that run on the Android OS have been considered but have not met at least one of the requirements, most notably:

Couchbase and Couchbase Mobile Couchbase is supposed to be a "simple, fast, elastic" NoSQL Database [5]. It also offers an Android version called Couchbase Mobile. Unfortunately Couchbase cannot be run in embedded mode, a separate server component has to be deployed and started. The communication between the application and the database server is done via HTTP. This communication overhead is simply too big for a mobile application.

HSQL DB HSQL is a database written in pure Java that supports the SQL92 standard [16]. Unfortunately HSQL does officially not support the Android OS. Application can access the Database through JDBC which creates some overhead. This biggest issue with JDBC are concurrent transactions. JDBC allows only one transaction per opened connection. Also connections are not thread-safe and must be synchronized when shared between different threads. The synchronization and connection pooling that would be required in order to support many concurrent transactions would add unnecessary complexity.

H2 Database H2 Database [7] is similar to HSQL. It is written in pure Java and is SQL92 compliant. H2 runs on both J2SE and (since recently) also on Android OS. Just like HSQL, H2 is based on JDBC with all its disadvantages described above.

Performance evaluation of Berkeley DB Java Edition and Android SQLite

In order to decide between SQLite and Berkeley DB a small performance benchmark was developed. It consisted of reading and writing key-value pairs. Keys were of type `long` while values were `byte[]` with a random length between 100 and 5000 and random content. The benchmark was run on a HTC Legend with Android 2.3.3 and the results were measured with the Traceview profiler [11]. The benchmark consisted of several parts:

- Writing 100 values with sequential keys into an empty container (SQL table in case of SQLite and a database container in case of Berkeley DB).
- Writing 100 values with random (but unique) keys into an empty container.
- Reading 25 values with random keys from a container which stored 100 values.
- Reading all values from a container from a container which stored 100 values.
- Inserting 10 values into a container which already stored 100 values.
- Removing all entries from a container which stored 100 values.

Figure 6.3 shows the results. All operations were carried out in a single thread. The Android built-in SQLite database was clearly faster than Berkeley DB.

SQLite on J2SE

Unfortunately it is not possible to use SQLite in both the Android and J2SE versions of MozartSpaces for two reasons:

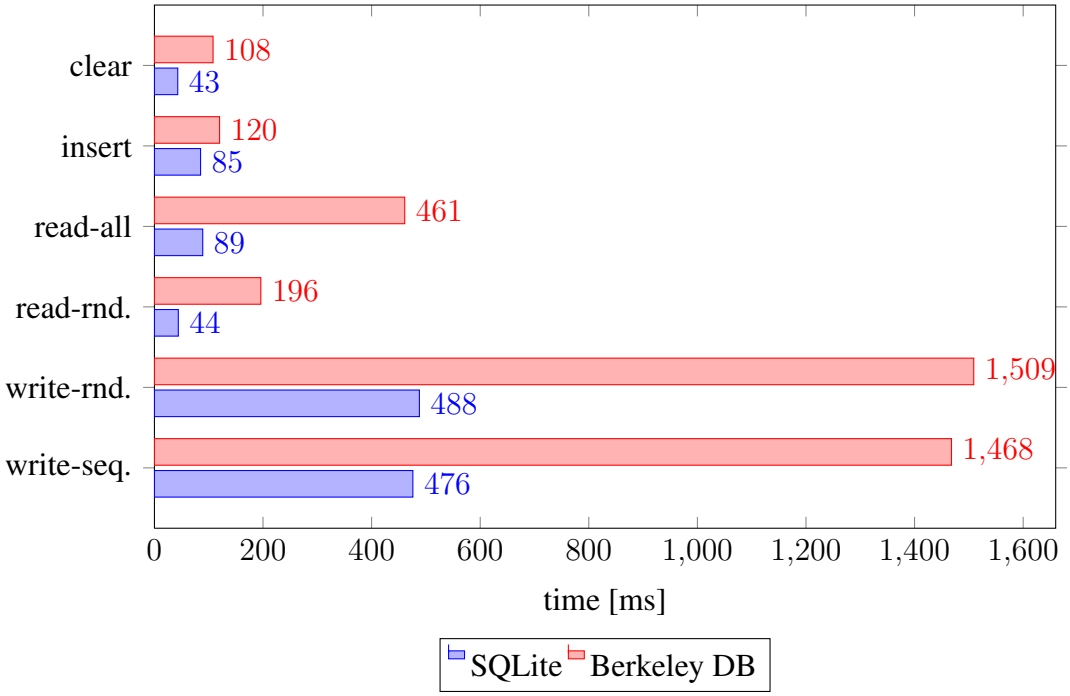


Figure 5.4: Performance evaluation of SQLite and Berkeley DB on Android

- First of all SQLite is written in C and compiled to native binaries. This means that a different precompiled binary has to be redistributed for each operating system and hardware architecture. This is not an issue on the Android version of MozartSpaces because SQLite is already present a part of the Android operating system. However with J2SE this is not the case which means that a OS- and hardware-specific version of SQLite would have to be redistributed with the J2SE version of MozartSpaces. This would make the build process and distribution of MozartSpaces more complicated and it would violate the requirements specified in previous section.

- The Android OS provides special Java language bindings for the SQLite (see package `android.database.sqlite` of the Android API). There are also Java language bindings for J2SE but with different API. This means that Java code which uses SQLite on Android cannot be compiled with J2SE because the API is different.

Conclusion

In the end it was decided to use Android SQLite database for the Android version of MozartSpaces mainly because of its performance and because it is available on all An-

droid devices and does not require any additional libraries.

Berkeley DB Java Edition was chosen as the default storage engine for the J2SE version of MozartSpaces. It is also be possible to use Berkeley DB on Android as well but that would mean additional 2MB of libraries and require external storage (e.g. SD card) as well as permissions to access it.

Support for other databases can be added by simply implementing an interface and configuring the MozartSpaces core. H2 Database might be supported in the future as an alternative to Berkeley DB.

5.3 Abstraction of the storage engine

The two selected storage engines use different database models. Berkeley DB is a key-value store that supports concurrent transactions. SQLite on the other hand is an SQL database and the version used on the Android operating system only supports one exclusive transaction at a time. The persistent layer must provide an abstraction of the two storage engines. It also must provide a way to implement a fast but not persistent in-memory mode which stores the data in Java collections.

The `StoredMap` interface

The `StoredMap` is generic key-value store that supports transactions and one of the central parts of the persistence. Table 5.1 shows all methods of the interface and their description. The names of the methods were chosen to be similar to the `java.util.Map` interface because it was intended to replace of `ConcurrentHashMap`s used by previous versions of MozartSpaces. By design the `StoredMap` does not provide transactional isolation. All read operations (`get(key)`, `containsKey(key)`, `size()`, `keySet()`) can read data written by both committed and uncommitted transactions. There are two types of write operations: with explicit transactions (`put(key, value, tx)` and `remove(key, tx)`) and with implicit transactions (`put(key, value)` and `remove(key)`). Effects of write operations with implicit transactions are persisted immediately. Effects of write operations with explicit transactions are persisted once the specified transaction is committed. See Section 5.4 for more details.

There are five classes that implement this interface:

- `InMemoryStoredMap` which is backed by a `ConcurrentHashMap`. This implementation keeps the data in memory only.
- `CachedKeySetStoredMapWrapper` is a wrapper for an arbitrary `StoredMap` that caches the set of keys stored in the map (see Section 5.5).

Method	Description
`get(key)`	returns the (possibly uncommitted) value associated with `key`
`put(key, value, tx)`	stores the new `value` associated with the given `key` in the map. The effects of operation will be made persistent once the transaction `tx` is committed.
`put(key, value)`	same as `put(key, value, tx)` except an implicit transaction is used which means that the effects of the operations are made persistent immediately.
`remove(key, tx)`	removes the given `key` and the value associated with it. The effects of the operation will be made persistent once the transaction `tx` is committed.
`remove(key)`	same as `remove(key)` except an implicit transaction is used which means that the effects of the operations are made persistent immediately.
`containsKey(key)`	returns `true` if the map contains the given `key`
`size()`	returns the number of key-value pairs stored in the map including uncommitted data
`keySet()`	returns an immutable set that contains all the keys currently stored in the map
`clear()`	removes all key-value pairs
`close()`	closes the map and releases all resources attached to it
`destroy()`	closes the map, deletes all contents and the map itself and releases all resources attached to it

Table 5.1: The `StoredMap` interface

- `AbstractStoredMap` is an abstract class that implements parts of the `StoredMap` interface. It converts the keys (which can be of arbitrary type) to `PersistenceKeys` and vice versa (see next section).
- `BufferedStoredMap` is backed by backends that do not support concurrent transactions. Because of this all write operations have to be first written to a buffer and executed later when the transaction is committed (see Section 5.4).
- `TransactionalStoredMap` is backed by backends that support concurrent transactions. This is done by mapping CAPI-transactions to the transactions of the backend (see Section 5.4).

The `PersistenceContext` class

The `PersistenceContext` class encapsulates a configured environment that can be used to persistently store data. It provides methods for managing `StoredMaps` and creating new persistent transactions. A persistence context is always using one profile (see Section 5.7) which uses one backend. Table 5.2 shows the public methods of the class.

Method	Description
`createStoredMap (name, persistentKeyFactory)`	creates a new `StoredMap`
`createPersistentTx (tx)`	creates a new `LogItem` that represents the persistent transaction and can be attached to the CAPI transaction.
`close()`	closes the `PersistenceContext` and releases all resources attached to it
`createOrderedLongSet (name)`	creates a new `OrderedLongSet` (see Section 5.8)
`makeEntryLazy (nativeEntry)`	returns a new `NativeEntry` that uses lazy loading of the actual entry value (see Section 5.8)

Table 5.2: The `PersistenceContext` class

The `PersistenceContext` is initialized at the startup of the space. The constructor takes the name of the persistence profile as parameter and based on this name resolves the appropriate backend and initializes it. The `PersistenceContext` is closed when the space is shut down to safely release all resources used by the backend.

The method `createStoredMap(name, persistentKeyFactory)` returns an instance of the `StoredMap` interface. The actual implementation class depends on the backend. In case of Berkeley DB or Android SQLite the returned map already contains the key-value pairs written in it during the previous sessions.

The `PersistenceKey` interface and the `PersistenceKeyFactory` interface

The `StoredMap` interface allows keys of arbitrary type just like all the implementations of the `Map` interface from the Java collections backend. This is important to keep changes in existing codebase of MozartSpaces at a minimum. The backends on the other hand pose some constraints on the keys. Berkeley DB stores and manages the keys in form of a `byte[]` while SQLite requires that the keys are one of the allowed SQLite

datatypes: INTEGER, REAL, TEXT, BLOB [30]. Any of these types can be used as a primary key but the access is about twice as fast if an INTEGER is used as primary key [32].

The `PersistenceKey` interface is an abstraction of the key type and wraps the actual key objects used by the `StoredMap`. It is used to convert the key to a form compatible with the database. It has a type parameter which specifies the actual type of the key. Table 5.3 shows the methods of the interface.

Method	**Description**
`isConvertibleToLong()`	returns `true` if the key can be converted to `long` value
`isConvertibleToString()`	returns `true` if the key can be converted to `String` value
`asString()`	returns the `String` representation of the key
`asLong()`	returns the `long` representation of the key
`asByteArray()`	returns the key as `byte[]`
`getKey()`	returns the key itself

Table 5.3: The `PersistenceKey` interface

The backend implementation decides which form of the key is used for storage. If Berkeley DB is used then the keys are always converted to `byte[]`. In case of Android SQLite the key is converted to `long` if possible (determined by the `isConvertibleToLong()` method), otherwise the `String` representation is used.

The reverse conversion (from a database value into a key of a particular type) is done with a `PersistenceKeyFactory`. This interface is also parametrized by the type of the actual keys. Table 5.4 shows the methods of the interface.

There are four different implementations of the `PersistenceKey` interface (along with the four matching `PersistenceKeyFactory`s):

- `LongPersistenceKey` which wraps a `long` value
- `NativeEntryPersistenceKey` which uses the ID of a `NativeEntry` (which is a `long`) as a value
- `StringPersistenceKey` which wraps a `String`. Naturally it is not convertible to `long` which can lead to slightly reduced performance in SQLite. Fortunately there are no performance-critical parts that use `StoredMap`s with `String`s as a key type.
- `ClassPersistenceKey` which wraps the name of the class (`String`). These keys are also not convertible to `long`.

Method	Description
`canConvertFromString()`	returns `true` if the factory can create a `PersistenceKey` from a `String` value
`canConvertFromLong()`	returns `true` if the factory can create a persistence key from a `long` value
`createPK(key)`	creates a new `PersistenceKey` from an arbitrary object
`createPKFromByteArray (data)`	creates a new `PersistenceKey` from a `byte[]`
`createPKFromLong (long)`	creates a new `PersistenceKey` from a `long`
`createPKFromString (string)`	creates a new `PersistenceKey` from a `String`

Table 5.4: The `PersistenceKeyFactory` interface

This means that currently only `long`s, `NativeEntry`s, `String`s and `Class`es can be used as keys in a `StoredMap`. Currently only `LongPersistenceKey` and `NativeEntryPersistenceKey` are being actively used in MozartSpaces. Support for other types can be added by implementing the `PersistenceKey` and `PersistenceKeyFactory` interfaces.

The `BaseDBAdapter` interface

Compared to a `StoredMap` database adapters provide a more low-level abstraction of a database container (e.g. an SQL table). A database adapter has database-specific code. Just like a `StoredMap` it is basically a key-value store but it processes the data on a lower level. Values written and retrieved by a database adapter are already in serialized form (`byte[]`).

The common functionality of all database adapters is represented with the `BaseDBAdapter` interface which is described in Table 5.5.

The `BaseDBAdapter` interface does not have any method for actually storing and retrieving any values. It only has general methods for initialization and data management. There are two interfaces that extend the interface.

The `DBAdapter` interface

The `DBAdapter` interface extends the `BaseDBAdapter` interface. Its methods are described in Table 5.6. It is intended for databases that do not support concurrent long running transactions. Effects of methods that modify the content (`put(key, data)`

Method	Description
`init(properties)`	initializes the adapter with the configuration provided in the `properties`
`destroy()`	closes the database adapter, removes all key-value pairs and the database container itself
`close()`	closes the database adapter and releases all resources
`clear()`	removes all key-value pairs
`count()`	returns number key-value pairs stored in the container managed by this adapter
`keySet()`	returns a set of `PersistenceKeys` stored in the container managed by this adapter

Table 5.5: The `BaseDBAdapter` interface

and `delete(key)`) are persisted immediately. The `BufferedStoredMap` (see Section) internally uses this adapter.

Method	Description
`get(key)`	returns the value (as `byte[]`) associated with the `key`
`put(key, data)`	store a new key-value pair and make it persistent immediately
`delete(key)`	delete a key-value pair with the given `key` and persist the change immediately

Table 5.6: The `DBAdapter` interface

The `AndroidSQLiteDBAdapter` is an implementation of this interface which stores the key-value pairs in an SQL table of the SQLite database on the Android operating system.

The `TransactionalDBAdapter` interface

The `TransactionalDBAdapter` interface as described in Table 5.7 is intended for database containers that support concurrent long running transactions. The `TransactionalStoredMap` (see Section 5.4) internally uses this adapter. The methods that modify the content of the database (`put(key, data, tx)` and `delete(key, tx)` have a parameter to specify the transaction in which the operation should be carried out. The effects of the operation are persisted when the transaction is committed.

The `BerkeleyDBAdapter` implements this interface by storing the key-value pairs in a Berkeley DB container.

Method	Description
`get(key)`	returns the value (as `byte[]`) associated with the `key`
`put(key, data, tx)`	store a new key-value pair and make it persistent when `tx` is committed
`delete(key, tx)`	delete a key-value pair with the given `key` and persist the change when `tx` is committed

Table 5.7: The `TransactionalDBAdapter` interface

5.4 Transactions

Integrating the persistence with the transactions of MozartSpaces was probably the most complex task. The main reason for this is the transactional model of the XVSM specification. XVSM supports concurrent long running transactions and unlike most databases uses pessimistic locking for transactional isolation. Two different persistent transaction implementations were created one of which is used depending on whether the persistence backend supports concurrent long running transactions or not. In the former case the so called buffered transactions (named after their internal buffers) are used and in the latter case the so called mapped transactions (since it is basically a 1-to-1 mapping between CAPI transactions and database transactions) are used.

Buffered transactions

The persistence layer uses buffered transactions if the database does not support concurrent long running transactions. In this case the transactional behavior has to be implemented by the persistence layer (and not the backend) which creates some overhead. The sequence diagram in Figure 5.5 shows a sample transaction that is started, executes three operations (put, get, remove) and is finally committed.

When a new buffered transaction is started a new `PersistentTransaction` is instantiated. It implements the `LogItem` interface and thus can be added to the transaction log of the CAPI transaction. `PersistentTransaction` acts as a container of write operations (put and remove). It is also added to the `DeferredDB` which is basically a container (a set) of `PersistentTransaction`s.

The `BufferedStoredMap` has two internal buffers that are important for the read-uncommitted functionality of the `StoredMap`. The write-buffer contains key-value pairs that have been written but not yet committed. The delete-buffer is similar and contains deleted key-value pairs. Both buffers are shared by all transactions and access to them must be synchronized.

Calling the put method of a `StoredMap` (in this case the `BufferedStoredMap` implementation to be precise) causes three things to happen. First the new key-value

pair is written into a write-buffer and second the write operation is added to the `PersistentTransaction`. Finally the new key-value pair is added to the database cache (see Section 5.5). If the key is already stored in the map then the old value is overwritten. Also the key is removed from the delete-buffer if present.

Reading from a `BufferedStoredMap` is done by first checking the write-buffer and the delete-buffer which contain data written/deleted by running (uncommitted) transactions. If the key is present in the delete-buffer then `null` is returned. If the key is present in the write-buffer then the value from the write-buffer is returned, otherwise the database cache is searched. Finally if the cache does not store the key then the database backend itself is queried.

Deleting a key-value pair from a `BufferedStoredMap` is similar to writing one. The key is written into the delete-buffer and deleted from the write-buffer if present.

When the CAPI transaction is committed all its `LogItems` are committed and among them also the `PersistentTransaction`. The operations stored in the buffers of that particular transaction are executed. Key-value pairs in the write-buffer are written into the database and keys in the delete-buffer are removed from the database. In case of SQLite this means running the INSERT and DELETE queries.

The transaction rollback is very simple. The key-value pairs in the write-buffer and the keys in the delete buffer of the particular transaction are simply removed from those buffers. The database itself remains unchanged.

Mapped transactions

Mapped transactions are used when the backend supports transactions with compatible semantics. This is the same approach as used in [Mei11]. The backend must allow concurrent long running transactions without isolation (read-uncommitted). Berkeley DB is currently the only supported backend that uses mapped transaction but there are also other databases that offer compatible transactions such as the H2 database.

The implementation is simpler than the buffered transactions. Figure 5.6 shows a transaction that contains several operation.

When a new CAPI transaction is created the backend also starts a new transaction that is attached to the CAPI transaction. There is a one-to-one relationship between CAPI transactions and backend transactions. The backend manages the mapping between the two groups of transactions. This is implemented with a `Map` where CAPI transactions are the keys and the backend transactions are the values.

When a CAPI transaction wants to write (or delete) some data the mapped backend transaction is used to execute the database operations. Since there is no isolation no backend transaction is needed for reading.

The backend is responsible for atomicity. The backend transaction is committed right before the CAPI transaction is committed and the same goes for rollback.

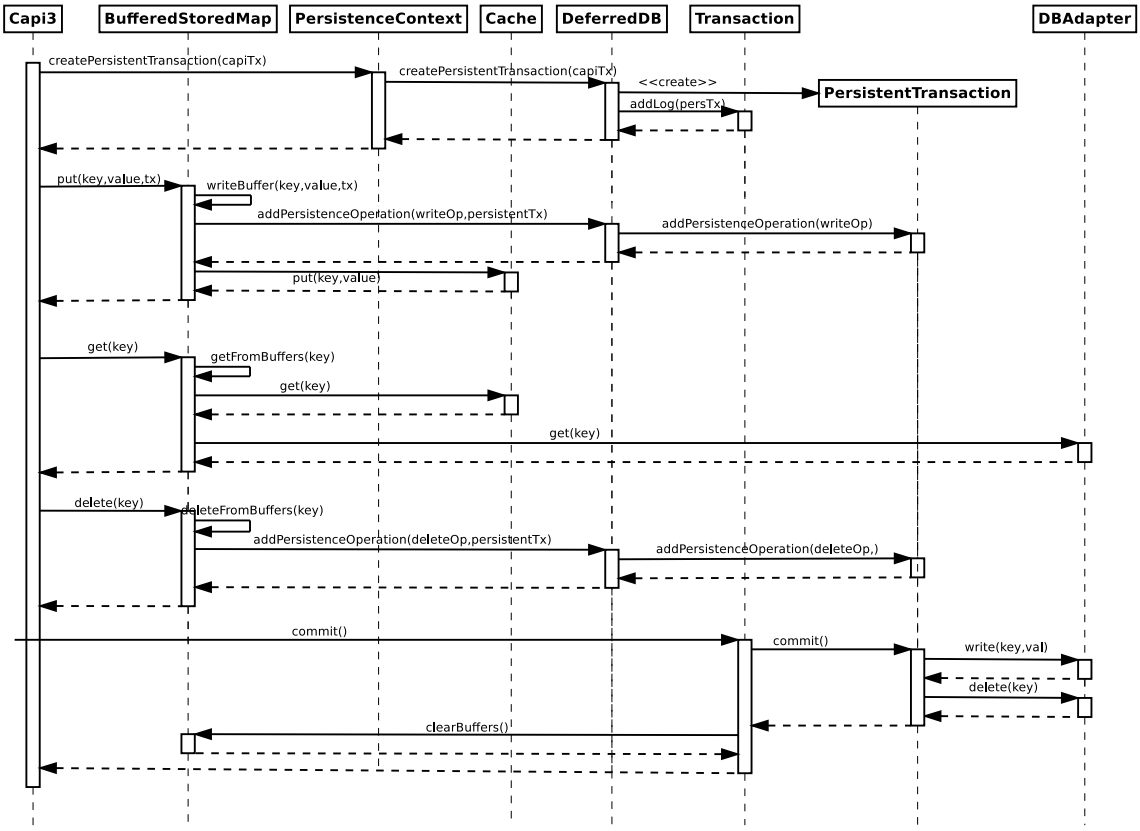


Figure 5.5: Sequence diagramm of `BufferedStoredMap`

5.5 Caching

Initial performance tests have shown that some parts of the database and some operations need caching. There are four important caches in the persistence layer.

Database cache

This is the cache for the values of the `StoredMap` and is represented by the `PersistenceCache` interface which is described in Table 5.8. There are two implementations of this interface. The default is `EmptyCache` which, as the name suggests, is empty and always returns null. This is used when no caching of database values is necessary. The other implementation is called `GuavaCache` and resides in the core-cache module that can be deployed optionally. It is based on the Guava library [12] and has a configurable size. Internally it uses a map of weak keys and weak values so the key-value pairs are automatically evicted by the garbage collector if there are no other references to them [25].

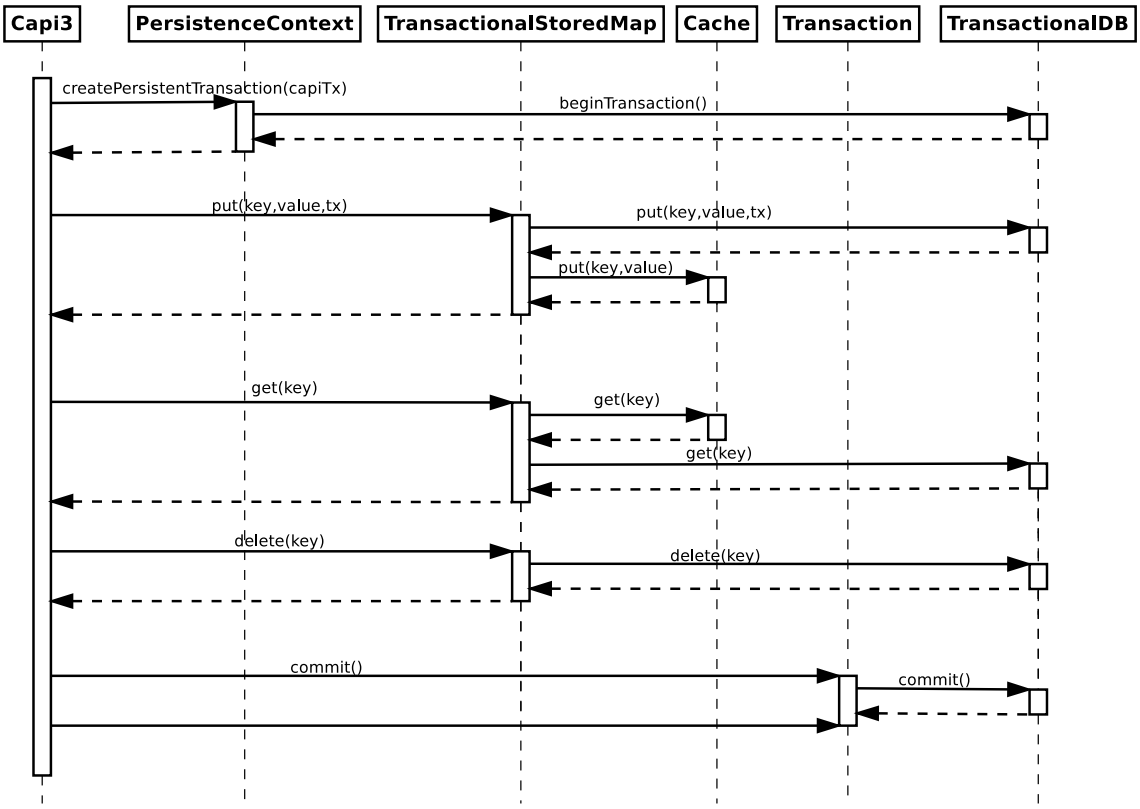


Figure 5.6: Sequence diagramm of `TransactionalStoredMap`

Key cache

During the processing of the request the value of an entry is often not important. Many coordinators operate only with the IDs of the entries. As a result the values from a `StoredMap` are not accessed very often. The keys on the other hand are needed very often. The set of keys that contains the IDs of all entries stored in a particular container is usually iterated several times during a request. Retrieving the keys from the database every time takes too much time, especially on Berkeley DB. Because of this and because the keys are usually very small (mostly just `longs`) it was decided to cache the set of keys in memory.

The caching of keys is done by the `CachedKeySetStoredMapWrapper` which implements the `StoredMap` interface. It is a wrapper that delegates all operations to the `StoredMap` it wraps. Additionally it keeps all keys of the `StoredMap` in a `ConcurrentHashMap`.

Serializer cache

All values need to be serialized before they are stored in the database. Depending on the data this can become a performance problem. The `CachingSerializerWrapper` offers a cache for serialization of objects. It uses the Guava library [12] and has a configurable size. As the name suggests it is a wrapper for the `Serializer` interface. The actual serialization process is delegated to the wrapped serializer. Additionally the object and the result (a `byte[]`) are stored in the cache. If the same object (according to the `equals` method) is to be serialized again the cached `byte[]`-value is returned.

Internal Berkeley DB cache

Berkeley DB uses an internal database to cache values. The size of the cache is configurable and when the limit is reached the values are evicted using the least recently used strategy.

Method	Description
`get(key)`	get a value from the cache associated with `key` or `null` if no cached value is found
`put(key, value)`	put a new key-value pair into the cache. It will remain there util it is evicted (by implementation-specific decision process) or until it is explicitly removed
`remove(key)`	explicitly remove the value associated with `key`

Table 5.8: The `PersistenceCache` interface

5.6 Configuration

The persistence layer uses the same configuration tools as the rest of MozartSpaces. In most cases the configuration is done via a configuration file in XML format. The structure of this file was extended and can now also contain configuration values for the persistence. Listing 5.1 shows a sample XML configuration file. Alternatively MozartSpaces can be configured programmatically using the `Configuration` class.

The configuration consists of the following parts:

- **Persistence profile** is a string which determines which persistence profile should be used. Currently available are “in-memory”, “lazy-berkeley”, “transactional-berkeley”, “transactional-sync-berkeley” and “android-sqlite”. It is also possible to specify a custom profile by providing the name of a class that implements the `PersistenceBackend` interface. The default is “in-memory”.

```
<mozartspacesCoreConfig>
  <!-- ... normal MozartSpaces config is here ... -->
  <persistence>
    <profile>transactional-berkeley</profile>
    <serializer>xstream-xml</serializer>
    <serializerCacheSize>5000</serializerCacheSize>
    <properties>
      <property key="berkeley-location">/tmp/</property>
    </properties>
  <persistence>
</mozartspacesCoreConfig>
```

Listing 5.1: Sample persistence configuration

- **Serializer** - The name of the serializer that should be used to convert the objects into `byte[]` before they are stored in the database. Possible values are "javabuiltin", "jaxb", "xstream-xml", "xstream-json", "kryo" or a name of a class that implements the `Serializer` interface. The default is "javabuiltin".
- **Serializer cache** - The number of objects cached by the `CachingSerializerWrapper` together with their `byte[]`-value. The default is 0 which means that no serialization cache is used.
- **Properties** - A list of key-value pairs that can be used to specify implementation-specific configuration details. Table 5.9 shows possible keys and their descriptions. The default is empty.

Persistence profile	**Property key**	**Description**	**Default value**
android-sqlite	sqlite-db-name	The name of the SQLite database	"mozart-spaces"
all *-berkeley profiles	berkeley-location	Path to the directory where database stores its data	<temp-directory> + "/xvsm"
all *-berkeley profiles	berkeley-cache-size	Size of the internal cache of Berkeley DB in bytes	10.000.000

Table 5.9: The persistence configuration properties

5.7 Persistence profiles

The persistence layer offers five different persistence profiles. Each profile has its advantages and disadvantages and is intended for a different application scenario. A brief overview of the five profiles is given in Table 5.10.

Berkeley DB profiles

The persistence profiles based on the Berkeley DB are available on both the J2SE version and the Android version of MozartSpaces.

Berkeley DB writes all operations into a database log in the file-system. This log can be used later to restore the state of the database. The location of the log can be configured. The sync-policy defines exactly how and when the log is written [22]. All three profiles are identical except for the sync-policy. This leads to different performance and semantics of the persistence.

All three profiles use the cache provided and managed automatically by Berkeley DB. The size of the cache can be configured.

Lazy profile

In this profile the database log is written asynchronously at some point in the future. Berkeley DB writes the operations into an in-memory log buffer and flushes them to a file when the buffer is full. This means that data of committed transactions can be lost if the system crashes after the transaction commit but before the log is flushed. The advantage of this profile is its good performance because it avoids many small write operations on the file system.

Transactional profile

In this profile the database log is written synchronously when the transaction is committed. However this does not mean that data is persistent because most file systems use some sort of buffer before actually writing the data to the hard drive. Data can be lost if a transaction has been committed, the database log was written but the file was not yet flushed to the hard drive.

This profile should provide a reasonable compromise between performance and data recoverability in case of a crash.

Transactional with fsync profile

In this profile the database log is written synchronously when the transaction is committed and the file which holds the database log is flushed to the hard drive. Of course there is a performance penalty associated with flushing the log on every commit. On the

other hand this provides true ACID properties. After a transaction has been committed it is guaranteed that the data is persistently stored.

Android SQLite profile

This profile is only available in the Android version of MozartSpaces. All data is stored in the embedded SQLite database provided by the Android OS. The system takes care of actually writing the SQL tables to a persistent storage.

In-memory profile

This is the default profile when persistence is not configured. It is available in both the J2SE and the Android version of MozartSpaces. All data is stored in-memory using the Java Collections Framework. The behavior of MozartSpaces with this profile is the same as it was before the persistence was implemented. This means that there is no persistence and all data is lost when the space is shut down or the JVM/DalvikVM terminates. The number of entries and containers that can be stored in the space is limited by the memory available to the JVM/DalvikVM. This profile provides the best performance because there is no need to write/read data from/to the persistent storage.

Name	**Lazy**	**Transactional**	**Transactional with fsync**	**Android SQLite**	**In-memory**
Storage engine	Berkeley DB	Berkeley DB	Berkeley DB	Android SQLite	Java Collections
Available on platform	J2SE and Android OS	J2SE and Android OS	J2SE and Android OS	Android OS	J2SE and Android OS
Intended for	High performance with non-critical database	Reasonable compromise between data security and performance	Critical data	Android applications with small data sets	When no persistence required and all data can fit into memory
ACID	No	Depends on file-system	Yes	Yes	No
All data in memory	No	No	No	No	Yes
Data written	Asynchronously at some point in future	Synchronously on commit	Synchronously on commit	Synchronously on commit	No
Loss of data possible	On application or system crash	On system crash	No	No	On application shutdown
Cache	Configurable size, managed by Berkeley DB	Configurable size, managed by Berkeley DB	Configurable size, managed by Berkeley DB	No	No

Table 5.10: Overview of persistence profiles

5.8 Changes in existing parts of MozartSpaces

This section describes the changes that had to be made to existing parts of MozartSpaces. Most of the changes are in the Java native implementation of CAPI-3. Only minor changes had to be made to the runtime and the XVSMP.

Changes to the runtime

Changes to the MozartSpaces Runtime were minimal. Some minor changes have been made to the initialization process. The `DefaultMzsCoreFactory` which is responsible for initialization of CAPI-3 has to read the configuration of the persistence and initialize the `PersistenceContext` accordingly. This is the so called "configured" `PersistenceContext`. Additionally there is a second `PersistenceContext` which always uses the "in-memory" profile.

An additional parameter was introduced in the `createContainer` method in `Capi` and `AsyncCapi` interfaces which are used by clients that want to access the space. This parameter specifies whether the newly created container should use the default persistence profile (as defined in the space configuration) or whether the container should be in-memory only without any persistence. Depending on the value of this parameter the "configured" or the "in-memory" `PersistenceContext` is used during the creation of the container. This new parameter made minor changes in the request and task classes necessary. The platform-independent XVSMP was also adapted.

Changes to containers

The `NativeContainer` interface and its implementation class `DefaultContainer` represent a single container at runtime. Several changes had to be made to support persistence. First of all the `ConcurrentHashMap` that used to hold all entries was replaced by a `StoredMap` with the entry IDs as keys and `NativeEntry`s as values.

The `DefaultContainer` is responsible for restoring the entries from the persistent storage and reregistering them with the coordinators (see Section 5.10).

Changes to the container manager

As the name suggests the `DefaultContainerManager` is responsible for managing containers. It is used to create, delete and look up containers. At runtime there is only one instance of this class that is initialized at the startup of the space.

The container manager now has a `StoredMap` that uses container IDs as keys and `PersistentContainerDescriptor`s as values. A `PersistentContainerDescriptor` contains all information required to recreate a container, namely:

- the container ID and name
- the container size (the maximal number of entries allowed in the container)
- two lists containing obligatory and optional coordinators

The descriptor does not store the contents of the container (the entries). Neither does it contain the internal state of the coordinators.

Changes to coordinators

The predefined coordinators in MozartSpaces are all based on the same principle. They all use one or more Java collections to store their state and manage the entries. When an entry is written into a container it gets registered with the associated coordinators. This means that the entry is added to the collections of entries managed by the coordinator together with the coordination data that was supplied as part of the write request (e.g. label in case of the Label coordinator). When the container is queried the coordinator searches its collections of entries for any matching results. When the entry is deleted it is also deleted from the collections used by the coordinator.

For performance reasons there is often a lot of redundant information kept by the coordinators. For example the Label coordinator, which allows to assign each entry a string label and then later retrieve it using that particular label, uses two different maps: one that stores the label for each entry and another one that stores a list of entries for a given label. This is necessary because sometimes entries are looked up by a label and sometimes the other way around. But it is not necessary to persist both maps since one can be reconstructed from the other.

All predefined coordinators were adapted to use the persistent storage. Only information that is needed to restore the last consistent state of a coordinator is persisted. The stored information is usually not used during the runtime since the Java collections offer better performance for writing, selecting and deleting of data.

Coordinators that use the persistence layer have to implement the `PersistentCoordinator` interface which are depicted in Table 5.11. This interface is important for the restoration process described in Section 5.10.

Each `PersistentCoordinator` must provide a `CoordinatorRestoreTask` which is a serializable object that can create a new empty instance of the given `PersistentCoordinator`. This is important because most coordinators do not provide a default constructor. These `CoordinatorRestoreTasks` are then stored in the `PersistentContainerDescriptor` of the enclosing container.

Method	Description
`preRestoreContent (persistenceContext)`	called during the restoration process after instantiation and before restoring the entries
`postRestoreContent (persistenceContext, nativeContainer)`	called during the restoration process after the container and its data are restored
`init (persistenceContext, nativeContainer)`	initialize the coordinator, called after a new container with the given coordinator is created
`close()`	close the coordinator and release all resources used by the persistence.
`destroy()`	delete all contents and release all resources used by the persistence. Called when the container is deleted.
`getRestoreTask()`	get the serializable task object to recreate coordinator in its empty state

Table 5.11: The `PersistentCoordinator` interface

Lazy entries

Often the coordinators do not really need the data stored in the entries. Many coordinators care only about the ordering of the entries (FIFO/LIFO, Vector) or the additional coordination data attached to the entries (Key/Label). Coordinators keep references to objects they manage in collections (maps, lists or sets). This means that entries and their data would never get garbage collected and reside in memory at all times.

To counter this coordinators operate on *lazy entries*. Until now an entry (implemented by the class `DefaultEntry`) contained an id, a reference to the container in which it was stored and a reference to the actual data of the entry (a serializable value). A lazy entry acts as a proxy [GHJV95] to the actual data. It holds only the id and the reference to the container. If the actual entry data is required it is loaded from the persistent storage and then cached in a weak reference [25]. This approach saves memory because the entry data is only loaded when really needed.

Label coordinator

The modified Label coordinator uses a `StoredMap` to store the label for each entry. This map is updated as new entries are written and old entries are deleted. During normal operations the reverse mapping (lookup of entries with a particular label) is more useful. This reversed mapping is implemented with `java.util.Map` because it needs to be fast.

When the space is shut down the mapping of labels to entries is lost since it is kept

only in memory. When the space is restored it is reconstructed using the data from the `StoredMap`.

Key coordinator

The Key coordinator works just like the Label coordinator. The only difference is that it uses keys which unlike labels have to be unique. This constraint has no effects in the persistence. A `StoredMap` is used to store the key for each entry.

FIFO coordinator

The FIFO coordinator uses a `StoredMap` to store the position of each entry in the queue. A counter is increased with each write operation. When an entry is added to the coordinator the current value of the counter and the entry are written into the `StoredMap`. When an entry is deleted it is also deleted form the `StoredMap`.

When coordinator is restored from the persistent state it iterates over the map and uses the positions to restore the queue (implemented with a `java.util.Deque`).

LIFO coordinator

The LIFO coordinator uses the persistence in the same way as the FIFO coordinator.

Vector coordinator

The vector coordinator posed a special challenge. Just like the FIFO and LIFO coordinator it uses the ordering of the entries. The coordinator uses a `java.util.Vector` to manage the entries. Storing the position of each entry in a `StoredMap` would be possible but not very fast. Vector coordinator allows entries to be written not only at the beginning or the end of the vector but also in the middle. If an entry is inserted or deleted in the middle of the vector the positions of all subsequent entries have to be shifted to the right or to the left. That would result in $O(n)$ write operations every time an entry is written or deleted. Since both Berkeley DB and SQLite use a binary tree to store the keys which requires $O(log(n))$ operations the resulting complexity would be $O(n * log(n))$ for every new or deleted entry.

To make inserting and deleting of entries fast the ordering is stored in a double linked list which is implemented on top of a `StoredMap`. Figure 5.7 illustrates the data structure. The IDs of the entries are used as keys of the map and the links to the previous and next entries as values. Accessing a certain index in such a list is expensive ($O(n)$) but accessing a certain key is cheap. Suppose you want to insert a new entry with the ID 15 in the middle (`index=2`). Using the in-memory vector you can see that the current entry at that index has ID 4 and the previous is 11. Inserting a new entry at

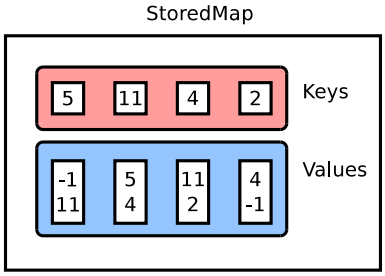


Figure 5.7: Doubly linked list in a `StoredMap`

the position means creating a new key value pair $(15, (11, 4))$ and updating the values for the entries 11 and 4 to $(4, 15)$ and $(15, 2)$.

Linda coordinator

The Linda Coordinator allows template-based queries as described in [Gel85]. In MozartSpaces the template matching is done with `LindaMatchers` which are class specific. This means that there is a matcher for each class that is used as the value of an entry. Given an instance a matcher can decide whether it matches the template or not by looking at instance variables. The matchers are held in memory only and have to be recreated when the space is restored. One way to do this would be to load and deserialize all entries from the persistent storage, look at the class of the value and then create the appropriate matcher. Of course this process would take very long. To speed it up the Linda coordinator uses a `StoredMap` to store the name of the class of each entry value because loading a class name (a `String`) from the storage engine is significantly faster than loading the value, deserializing it and then looking its class.

Query coordinator, Any coordinator and Random coordinator

These three coordinators do not require any additional data or ordering of the entries. The only information they store is the list of entries they manage. This is done by putting the entries into a `StoredMap`.

5.9 Initialization of the space and the persistence

This section describes how the is initialized at startup while focusing on the steps that are relevant for the persistence implementation. The startup of an embedded space is initialized by creating a new MozartSpaces core with the `DefaultMzsCoreFactory` and creating a new `Capi` instance which represents the CAPI-4 layer.

1. First the `DefaultMzsCoreFactory` loads the configuration – either from the supplied `Configuration` object or from the `mozartspaces.xml` file. The configuration contains information about the selected profile (Section 5.7), the caches (Section 5.5) and the backend-specific configuration properties.

2. Based on the configuration the `Serializer` used by the persistence is resolved and initialized. If a serialization cache is configured and the cache module is present then the `Serializer` is additionally wrapped with the `CachingSerializerWrapper`.

3. The persistence profile defined in the configuration is used to initialized the `PersistenceContext`. This is the so called "configured" `PersistenceContext`. Based on the profile the correct `PersistenceBackend` is resolved and initialized.

4. After the `PersistenceContext` is initialized the `DefaultCapi3Native` can be created. This is the implementation of the CAPI-1 to CAPI-3 layers.

5. The `DefaultCapi3Native` has two instances of `PersistenceContext`. The first is the "configured" one. Additionally a second "in-memory" `PersistenceContext` is created which is used for containers that do not require persistence.

6. Finally the `DefaultContainerManager` is created.

After these six steps the space and its persistence implementation are completely initialized but empty. Now the space is ready to restore all data that was saved by the persistence layer during previous execution.

5.10 Restoring the state of a space from persistent storage

At runtime MozartSpaces has a complex in-memory state which can now be stored persistently. When the space is shut down (or when it crashes) the persisted information should be sufficient to restore the state. The restoration process happens at the startup of the space after `PersistenceContext` has been initialized and the container manager was created.

This process has several steps:

1. First a new CAPI transaction is started in which the whole restoration process runs.

2. Then the containers have to be restored. This is done by the `DefaultContainerManager`. It iterates over the `PersistentContainerDescriptors` and processes one container at a time.

3. In order to restore the container the constructor of `DefaultContainer` has to be called. This constructor requires the container name, ID, size and list of obligatory and optional coordinators. This means that the coordinators already have to be instantiated when the container in constructed.

   Coordinators that implement the `PersistentCoordinator` interface can be constructed by the previously described `CoordinatorRestoreTask`. The remaining coordinators are built using the default constructor.

4. Once the coordinators are instantiated the `preRestoreContent()` method of all `PersistentCoordinators` is called. This allows coordinator-specific initialization before the coordinator is re-filled with the restored entries.

5. When the coordinators are instantiated and initialized the constructor of the container can be called.

6. Once the container is instantiated it loads all its entries and restores the value of the ID counter (an `AtomicLong`) that assigns IDs to new entries.

7. Then the locks of all the entries are recreated in the `IsolationManager`.

8. Now the `postRestoreContent()` method of all `PersistentCoordinators` can be called. At this point the coordinators can access the restored entries and recreate their last state based on the information saved in their private `StoredMaps`.

9. Finally the transaction of the restoration process is committed and the space is now up and running.

The runtime will not accept any requests until the restoration process is completed.

5.11 Optimization of performance and energy efficiency

The initial version of the persistence layer was neither fast not energy efficient. The initial runs of the benchmarks (see Chapter 6) had very disappointing results. The solution described in this chapter is the result of an optimization process.

Optimization of performance

The original design of the persistence layer did not provide the desired performance. Different profilers were used to identify the performance bottlenecks.

There are several profilers for J2SE applications available on the market. VisualVM [24] was used because it offers both CPU and memory profiling and because of its license.

J2SE profilers cannot be used for Android applications because of the differences in the architecture of the virtual machine. Android SDK comes with its own profiler called Traceview [11]. This profiler can be controlled from within the application code by using the Android debugging API. This is useful if only parts of the code need to be measured. In general code sections that created performance bottlenecks in J2SE were also problematic on the Android OS.

Optimization of energy efficiency

Optimizing energy efficiency turned out to be much more difficult than the optimization of performance. Unfortunately Android OS and its SDK do not offer any tools for direct measurement of how much power is consumed by a particular application. There are not many third party tools either.

PowerTutor [34] is a tool developed at the University of Michigan. It collects information about power consumed by different components of a mobile device, currently supported are CPU, various wireless network interfaces, screen, GPS sensor and audio system on mobile phones. The information about power consumption can be shown on a per-application basis. PowerTutor uses a model to estimate the power consumption of the hardware components [ZTQ+10]. This model was calibrated for the the HTC G1, HTC G2 and Nexus one mobile phones but it also works with reasonable precision for other devices [34]. Figure 5.8 shows PowerTutor running the XVSM energy efficiency benchmark.

Another tool that turned out to be very helpful is Trepn™Profiler [17] which is provided by Qualcomm, the manufacturer of CPUs that are used in many mobile devices including the Android mobile phone that was used during the development of the persistence (HTC Legend). Depending on the device compatibility Trepn Profiler can collect various information about the state of the CPU and the memory. Unfortunately Trepn Profiler was very unstable on the tested device and only provided information about the current drawn by the CPU and the overall electric charge usage of the CPU in mAh. Assuming that the voltage of the CPU remains constant over time the current in mA can be seen as power and the charge in mAh as energy consumed by the CPU.

As you can see in Figure 5.9 the energy usage of the CPU nicely copies the CPU load. Unfortunately neither PowerTutor nor Trepn Profiler was able to measure energy consumption of RAM and storage (internal flash memory and removable memory card)

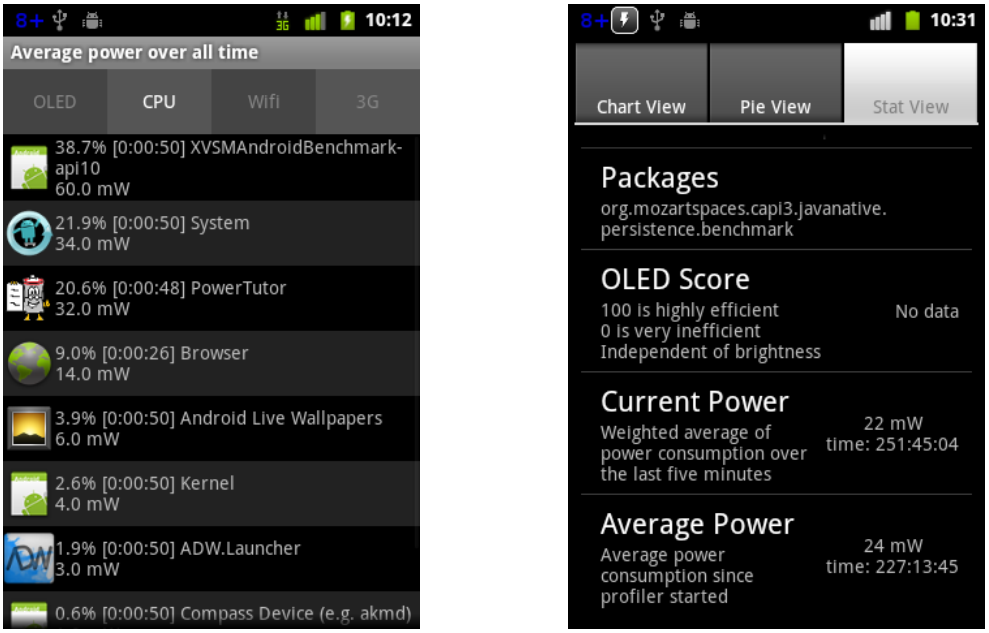
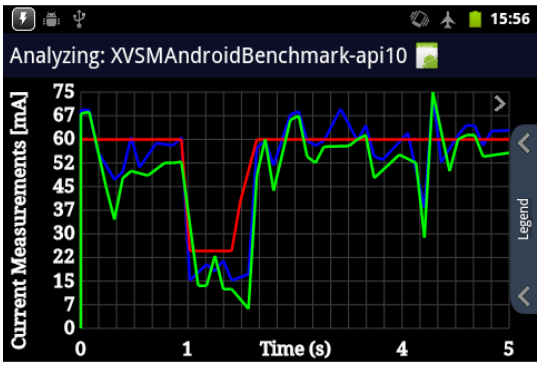


Figure 5.8: PowerTutor showing current energy usage per application (left) and per hardware component (right)

but according to [CH10] the CPU consumes significantly more energy than RAM and storage devices. Other hardware components such as the display and wireless connectivity are not directly used by MozartSpaces (at least not by the persistence) and therefore their energy usage needs not to be analyzed and optimized.

Developing energy efficient applications on Android OS turned out to be quite challenging because the developer has no direct control of the power consumption. The operating system provides an abstraction layer for the hardware components and the DalvikVM on top of it creates an abstraction of the whole hardware architecture. This allows development of portable application. On the other hand application developers have no direct control of the hardware resources and the energy they consume.



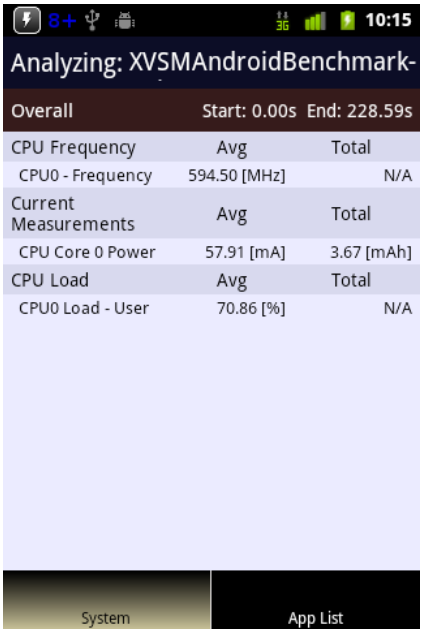


Figure 5.9: On the left: Trepn Profiler showing the CPU load (green), CPU frequency (red) and CPU current (blue), on the right Trepn Profiler showing total and average power consumption

CHAPTER 6

Evaluation

A series of benchmarks was created to measure the performance and energy efficiency of the implemented persistence layer. These benchmarks use the CAPI3 layer directly without the runtime and XVSMP. This eliminates the overhead created by the serialization and deserialization of requests, network transport etc. and measures directly the Java native implementation of CAPI3.

6.1 Benchmark environment

J2SE version

The performance of the J2SE version was measured on a notebook with an Intel Core 2 Duo running at 2.6GHz (P8600) with 4GB of RAM running the 64bit version of Ubuntu Linux 11.04 operating system. Java HotSpot™64-Bit Server VM version 1.7.0_02-b13 from Oracle was used as JVM.

Android OS version

The evaluation of the performance of the Android version was done on the European version of HTC Legend mobile phone with an Qualcomm CPU at 600MHz (ARM architecture) and 384MB of RAM running CyanogenMod 7.1 (Android 2.3.3).

6.2 Performance benchmarks

The performance benchmarks measured different parts of the implementation under different circumstances. Each of the benchmarks was executed with different number of

threads to measure the parallel scalability. The benchmarks were run with one, two, five and ten concurrent threads. Unless stated otherwise all results in this chapter are from runs with five concurrent threads.

In order to avoid performance impacts of the Java just-in-time compiler each benchmark was run 12 times and the fastest and the slowest times were discarded in order to eliminate outliers which can be caused by the just-in-time compiler or the garbage collector. The arithmetic mean of the remaining ten values was used as result.

Where applicable the benchmarks were run in different transactional settings:

- All operations were capsuled in one transaction per thread (named "single Tx" in the results).
- Each operation was run in a new separate transaction ("separate Tx")
- A new transaction was started after running 50 operations ("multiple Tx")

The benchmarks covered different operations listed bellow:

- writing 20,000 entries into a container with the AnyCoordinator
- reading all entries 1,000 times[1] and reading 50,000 entries each in a single operation from a container prefilled with 2,000 entries with the AnyCoordinator
- taking one, several or all entries at a time from a container prefilled with 10,000 entries with the AnyCoordinator
- writing 10,000 entries, reading entries (all in one step and 10,000 entries one at a time) as well as taking entries (10,000 - one at a time and 2,000 in one operation) from a container prefilled with 100,000 entries with the FIFO coordinator
- writing 10,000 entries, reading and taking entries from a container prefilled with 100,000 entries with the Key coordinator
- writing – appending 1,000 entries to the end of the vector and inserting 2,000 entries at `index=2` (thus shifting all subsequent entries), reading and taking all entries from a container prefilled with 10,000 entries with the Vector coordinator.
- writing 1,000 entries, reading and taking all entries from a container prefilled with 10,000 entries with the Linda coordinator

Benchmarks on the Android OS used numbers 200 times smaller to compensate the slower hardware. The other coordinators were omitted from the benchmarks because their implementation is practically identical to one of the four coordinators above. A detailed description of each benchmark can be found in Appendix A.

[1] 10 times on Android

Performance of the J2SE version

The performance of the J2SE version of MozartSpaces was measured in five different configurations:

1. a version of MozartSpaces without the persistence layer – 2.1-SNAPSHOT build 11418 from 02/07/2012 which from now on will be referred to as the "old" version of MozartSpaces. The performance of this version was used as base value, the performance values of the other four configurations in Table 6.1 are relative to this one.

2. the version of MozartSpaces with persistence as presented in this thesis (from now on referred to as the "new" version) with the in-memory persistence profile.

3. the new version of MozartSpaces with the lazy persistence profile based on Berkeley DB which was configured with 50MB of cache.

4. the new version of MozartSpaces with the transactional persistence profile based on Berkeley DB which was configured with 50MB of cache.

5. the new version of MozartSpaces with the transactional with fsync persistence profile based on Berkeley DB which was configured with 50MB of cache.

Figure 6.1 and Table 6.1 show the results of the performance benchmarks with five parallel threads.

The changes made to the architecture of MozartSpaces have some consequences for the performance. The in-memory profile is generally slower than the original version of MozartSpaces. The difference is not big but it is measurable. This is caused by the additional overhead created by the abstraction layer that is necessary for the persistence. In most cases the write operations need 20-50% more time and the performance of reading operations is more or less unchanged.

As expected the profiles that use Berkeley DB to store data persistently are significantly slower. This is the price for having persistent data. On the other hand these profiles use less memory since the data is not always stored in memory. Surprisingly there is little difference between the lazy and the transactional profiles. In general the write operations are 10-20 times slower. The performance of the read operations depends on the transactional setting. Reading many entries within one transaction is only slightly slower than the in-memory profile. Reading many entries each in its separate transaction is very slow (about ten times slower) because of the overhead caused by starting and committing many transactions.

In case of the Linda coordinator the overhead created by the persistence is barely measurable because most of the time is spent with the template matching.

The benchmarks have also shown that the persistence profile transactional with fsync is not usable for applications that require high transactional throughput. The price of

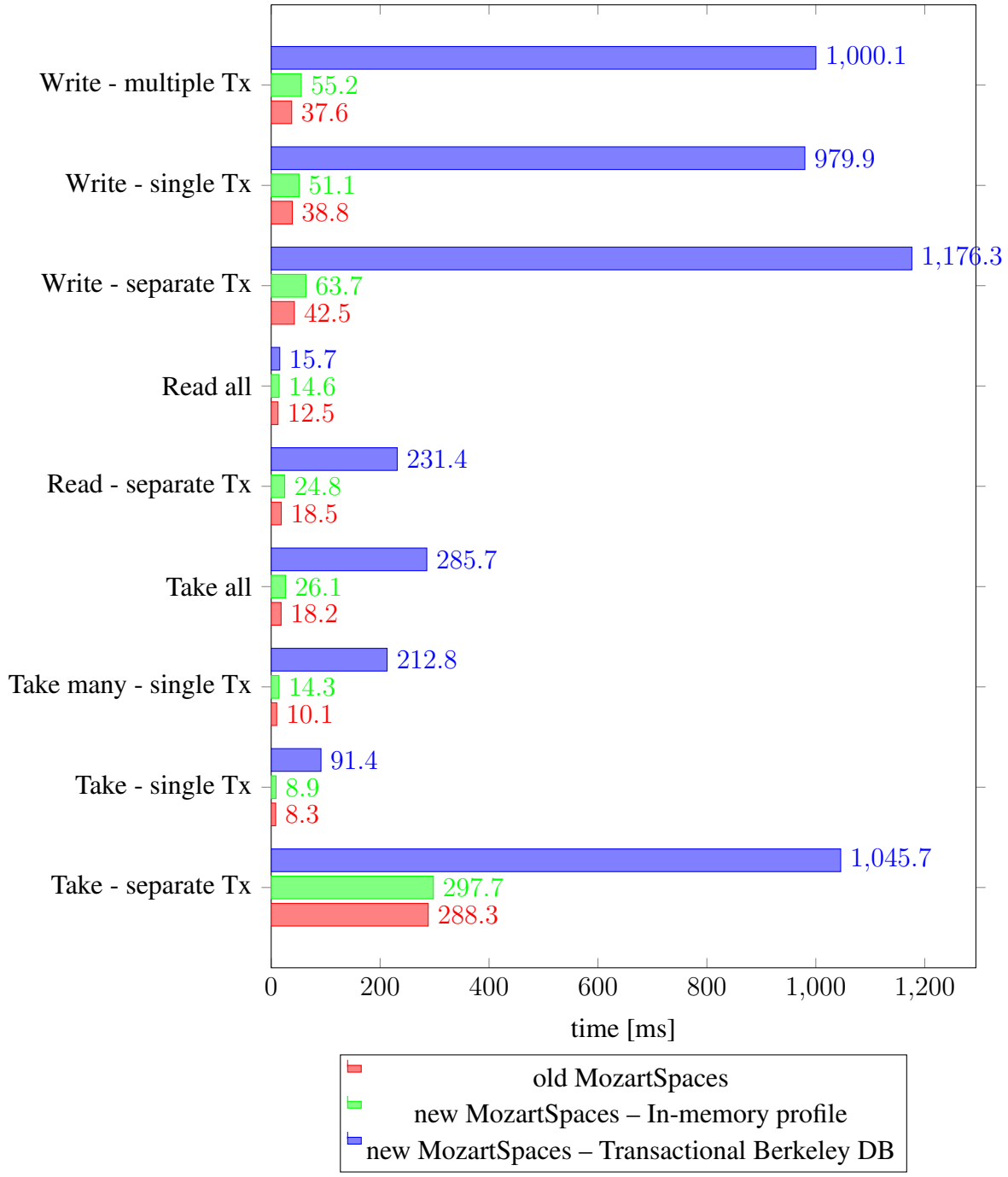


Figure 6.1: Performance evaluation MozartSpaces persistence with the AnyCoordinator with different profiles on J2SE

Bechmark	In-memory	Lazy	Transactional	Transactional with fsync
Any write - multiple Tx	147%	2603%	2660%	15860%
Any write - single Tx	132%	2488%	2526%	2647%
Any write - separate Tx	150%	2561%	2768%	386265%
Any read all	110%	186%	177%	155%
Any read - separate Tx	139%	1099%	691%	1153%
Any take all	138%	1686%	1514%	1831%
Any take many - single Tx	145%	2717%	2602%	3083%
Any take - single Tx	111%	186%	178%	171%
Any take - separate Tx	105%	382%	393%	18152%
Fifo add - single Tx	115%	1610%	1607%	2104%
Fifo read all	109%	232%	218%	205%
Fifo read - single Tx	96%	99%	97%	95%
Fifo take many - single Tx	132%	420%	516%	495%
Fifo take - single Tx	110%	177%	165%	297%
Key add - single Tx	115%	1387%	1396%	1771%
Key read - single Tx	118%	115%	109%	139%
Key take - single Tx	97%	110%	96%	113%
Vector add - single Tx	107%	813%	1012%	165114%
Vector append - single Tx	98%	522%	727%	147419%
Vector read - single Tx	111%	161%	139%	894%
Vector take - single Tx	88%	714%	899%	209904%
Linda write - single Tx	104%	120%	114%	117%
Linda read - single Tx	101%	105%	104%	103%
Linda take - single Tx	100%	110%	104%	106%

Table 6.1: Results of the performance benchmarks run with 5 parallel threads - J2SE version (relative to previous MozartSpaces version)

file system sync after each transaction commit is simply too high. This overhead is visible especially in benchmarks that use many transactions with just one operation. On the other hand effects on benchmarks that use one big transaction were relatively small. A fast storage device such a solid state drive could improve the performance but unfortunately no suitable hardware was available during the evaluation.

Performance of the J2SE version under different JVMs

The benchmarks were also reused to measure the influence of the JVM itself on the performance. The Oracle Java HotSpot™64-Bit Server VM version 1.7.0_02-b13 was

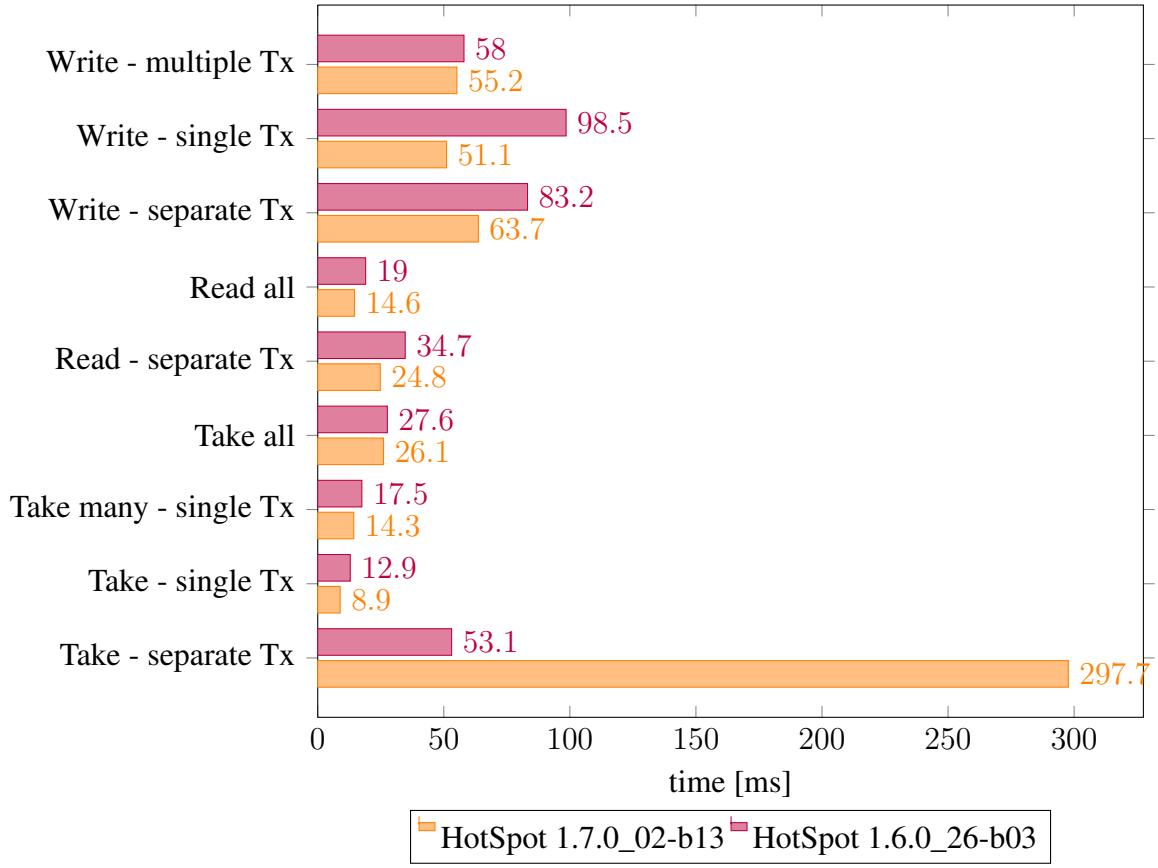


Figure 6.2: Performance evaluation MozartSpaces persistence with the AnyCoordinator with the in-memory profile with different versions of the Oracle HotSpot JVM

compared to version 1.6.0_26-b03.

The results are shown in the Figure 6.2. The version of the JVM had a significant impact on the performance. The newer version of the JVM caused MozartSpaces with in-memory profile to run 10-15% faster with one exception (take operations each in a separate transaction) which run four times slower. This was caused by the changes made in the virtual machine and the standard libraries that come with it [4]. The same comparison was also made with Berkeley DB profiles but the difference between the two JVM versions was barely measurable.

Parallel performance of the J2SE version

In most cases the performance of the Berkeley DB based profiles remained the same or got better when the number of entries writen/read/taken by the benchmark was divided across several threads. Especially I/O-intesive operations such as writing into a

container with a VectorCooridnator or taking entries got up to 50% faster with ten concurrent threads when compared to a single thread. This can be explained by better CPU utilization – while one thread is waiting for the I/O to finish another concurrent thread can use the CPU.

The only situation where the benchmarks got significantly slower with increasing number of threads was reading entries from a container with a FifoCoordinator.

Performance of the Android OS version

The performance evaluation of the Android OS version of MozartSpaces compared the in-memory profile, the SQLite profile and the transactional Berkeley DB profile. As you can see in Figure 6.3 both SQLite and Berkeley DB perform very well. In most cases both reading and writing takes about ten times as much time. The benchmarks in Section 5.2 suggested that SQLite is faster than Berkeley DB but now the opposite seems to be the case. Berkeley DB was faster in all Benchmarks except the first two. This is probably caused by better handling of concurrent transactions by Berkeley DB which was not tested in the benchmarks in Section 5.2.

Berkeley DB is not part of the standard distribution of MozartSpaces for Android but it can be deployed optionally.

6.3 Memory usage

Storing the data in the database also helped to reduce the memory usage of MozartSpaces at runtime. A simple benchmark was developed to evaluate the memory usage. Entries with random binary content were written into a container with a FiFoCoordinator and the heap size of the JVM was measured. The benchmark was carried out twice: with small entries (`byte[100]`) and with larger entries (`byte[10000]`). The same environment was used as in the J2SE performance benchmarks described above.

Figure 6.4 shows the results of the memory usage benchmark. There seems to be only a small difference between the in-memory profile and Berkeley DB when the entries are small. In fact if the entries are small then the in-memory profile uses less memory because the database creates some overhead with its internal bookkeeping. However the benchmark with the larger entries clearly shows the advantage of having a database. After an initial steep increase (caused by the 10MB internal database cache) the memory usage of Berkeley DB remains very flat while the memory usage of the in-memory profile grows rapidly.

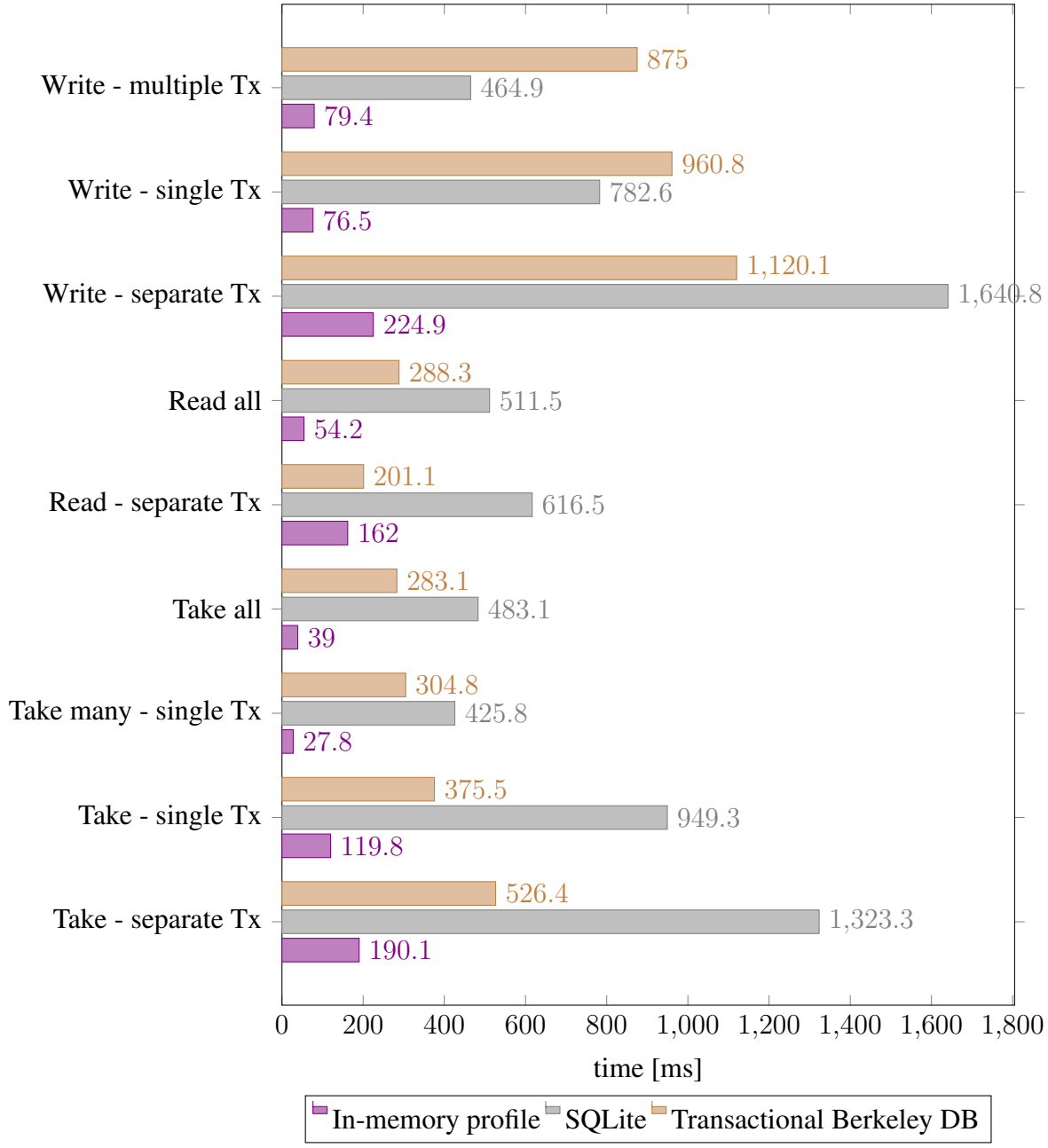


Figure 6.3: Performance evaluation MozartSpaces persistence with the AnyCoordinator with different profiles on Android OS

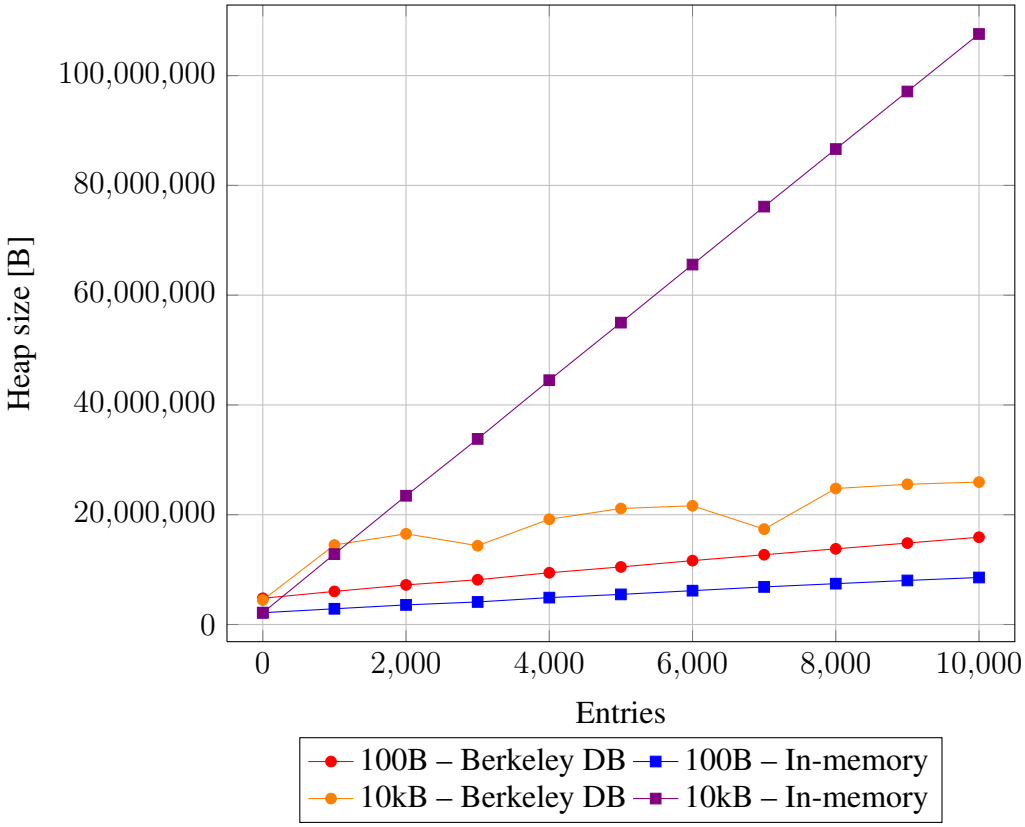


Figure 6.4: Memory usage of MozartSpaces with the In-memory profile and with the Transactional Berkeley DB profile

6.4 Energy efficiency

The goal of the energy efficiency benchmark is to evalute the costs of persistence in terms of electric power. Android OS is designed for mobile devices that often rely on battery power where energy efficiency is an important requirement.

Environment

The energy efficiency benchmark was run on the same device as the performance benchmark. Before each start the device was fully charged. In order to avoid side effects of network connections the SIM card was removed and the phone was put into airplane mode which disables all wireless radios (Wi-Fi, Bluetooth and cell network). The screen was switched off for the entire duration of the benchmark.

	In-Memory		SQLite		SQLite (relative)	
	Charge	Time	Charge	Time	Charge	Time
1st run	0.64mAh	33087ms	3.52mAh	215592ms	550%	652%
2nd run	0.64mAh	32128ms	3mAh	189055ms	469%	588%
3rd run	0.71mAh	34340ms	3.67mAh	201008ms	517%	585%
Average	0.663mAh	33185ms	3.397mAh	201885ms	512%	608%

Table 6.2: Results of the energy efficiency benchmarks using Trepn Profiler. The charge can be used as a measure of energy assuming that the CPU uses a constant voltage.

Design of the benchmark

The enery efficiency benchmark reused the code of the performance benchmark. Unfortunately it is not possible to observe the power consumption on a small scale to measure each benchmark separately. The numbers presented in this section were obtained by running the entire benchmark suite. This simulates heavy load caused by various space operations and transactions.

Results of the energy efficiency benchmark

The energy efficiency was measured with PowerTutor [34] and Trepn™Profiler [17] (see Section 5.11). PowerTutor provides data about energy usage of the CPU in Joule while Trepn Profiler the CPU power usage in mAh.

Tables 6.2 and 6.2 show the results of the energy efficiency benchmark. The tables also show the time it took to execute the benchmarks. The execution of the benchmark with SQLite took about six times longer and consumed about five to six times (depending on the profiler) more energy by the CPU.

The values measured by PowerTutor indicate that the benchmark causes the CPU to consume about 140-150mW on average. This goes for both the in-memory profile as well as the SQLite profile. To put these numbers into perspective the OLED display of the smartphone used for the benchmarks draws 300-500mW depending on the brightness, 3G mobile network draws 400-900mW, WiFi draws around 400mW and the sound chip 100mW (all values measured by PowerTutor). These numbers are similar to those published in [CH10].

Unfortunately it was not possible to measure the energy used by RAM and the internal flash memory where the SQLite database stores the data.

It is important to note that both profilers created a certain performance overhead. In case of the in-memory profile PowerTutor the benchmark execution took about 10% longer and Trepn Profiler slowed the benchmark down by ca. 50%. The overhead was about half as big (both profilers) when the SQLite profile was used.

	In-Memory		**SQLite**		**SQLite (relative)**	
	Energy	Time	Energy	Time	Energy	Time
1st run	3.5J	26827ms	21.5J	161730ms	614%	603%
2nd run	3.4J	22755ms	21.3J	158827ms	626%	698%
3rd run	3.4J	21340ms	21.9J	146488ms	644%	686%
Average	3.433J	23641ms	21.567J	155682ms	628%	659%

Table 6.3: Results of the energy efficiency benchmarks using PowerTutor

6.5 Summary

In general the performance of the persistence is very good. Naturally the persistence cannot be as fast as s purely in-memory solution. The decision to use Berkeley DB and SQLite as database engines turned out to be correct. Berkeley DB is fast on both platforms except for the Transactional with fsync profile but it is hardly surprising that flushing file system buffers after each transaction commit would cause performance degradation. SQLite might not be as fast as Berkeley DB but it is nicely integrated into the Android operating system.

Lowering the memory usage is also an important achievement. Applications that use large entries will benefit greatly from using a database.

It is unfortunate that the power profilers did not allow any optimizations but in general the energy usage caused by the persistence is quite small when compared to other hardware components even when under heavy load.

CHAPTER 7

Future work

XVSM and its reference implementation offer a unique set of features. The addition of a fast persistence layer is an important milestone for the project. However some problems still remain unsolved.

7.1 Possible improvements of the persistence layer

The performance, scalability and reliability of the persistence layer could be improved by using alternative database backends. The support for new backends can be added easily as plugins without any changes to the existing code.

Berkeley DB can be replicated across several nodes to improve availability and scalability [26]. This feature currently cannot be used by MozartSpaces because of the way how the database is configured and started. An alternative persistence backend could be implemented to provide a high level of availability. Of course this can be also achieved with other databases.

There are countless database systems available on the market each of them having its own unique set of advantages and disadvantages. The current architecture of the persistence layer requires a new backend binding for each database. The number of supported databases could be significantly increased by providing a more generic binding such as JDBC [23].

Storing entries by serializing them and writing them into database is very generic and allows entries of arbitrary type (as long as they are serializable). But this approach has some performance drawbacks because entries cannot be queried if they have the form of a `byte[]`. Selecting entries with the XVSM Query Language is very slow because the matcher needs to retrieve each entry value from the database and deserialize it before evaluating the query. This is the price for not having all entry values in memory

at all times. Increasing the cache size improves the performance but in also requires more memory at runtime. There are two options how this performance issue could be addressed:

- One possibility would be to implement more advanced caching. Right now the persistence layer caches the values using the last-recently-used strategy and treating all values equally. However entries that are accessed with XVSM Queries benefit from caching much more than for example entries managed by a KeyCoordinator. Allowing container-specific or coordinator-specific cache configuration could increase the performance of XVSM Queries without significantly increasing memory consumption.

- Another possibility would be to create indices for certain fields of entries that a queried. Usually not all fields are queried so not all data of an entry would have to be indexed. An in-memory index would make evaluation of queries possible without having to retrieve the entire entry value from database.

It is important to note that the second approach would be much more complicated because it would require a major rewrite of the query evaluation algorithm. It would however be more effective than simply caching many potentially large entries in memory as described in the first approach.

Another issue of the persistence layer that needs to be addressed are aspects. Currently there is no way for a client to get a reference to the instance of the `PersistenceContext` class during the execution of the aspect code. This means that aspects currently cannot use the persistence. Furthermore the persistence layer has no public API which could be used by the clients because it is part of the Java native implementation of CAPI-3.

Currently the decision whether a container will use persistence or whether it will be held in memory has to be done at the moment of the creation of the container. It is not possible to move an in-memory container to persistent storage once it has been initialized but sometimes this would be desirable. The biggest disadvantage of in-memory container is that they consume too much operating memory. In case of MozartSpaces this can lead to a `java.lang.OutOfMemoryError` because the JVM heap-space will become too big. A nice feature would be automatic swapping of in-memory containers. If an in-memory container becomes too big it would be automatically moved to persistent storage.

Also the persistence currently lacks maintenance and administration tools. For example is not possible to monitor how much storage each container is using. A tool for modifying, renaming, cloning, deleting of containers could also be very useful for large production deployments.

7.2 Other open issues of XVSM and MozartSpaces

The probably biggest issue that MozartSpaces currently faces is network connectivity. Most clients (e.g. mobile devices) do not allow or are not able to listen to incoming connections. This is normally caused by the network configuration (e.g. firewalls, network address translation). This makes asynchronous communication problematic. The matter is even worse with mobile devices because they usually do not have a fixed address. A mobile phone can change its address several times during a single day as it moves from one wireless/cellular network to another. MozartSpaces currently uses sockets for remote communication which is a very low-level approach. An alternative transport protocol based on technology that can handle different network configurations transparently would make deployment of truly distributed and decentralized applications based on MozartSpaces possible.

One of the possibilities being currently discussed by the XVSM technical board is using the Extensible Messaging and Presence Protocol (XMPP) [29]. It is very flexible and extensible. XMPP does require a server but there are many XMPP providers available and setting up an own XMPP server is very easy.

IPv6 [8] might also solve some aspects of the networking problem because it eliminates the need for network address translation (NAT). Unfortunately the adoption rate of IPv6 among network and internet providers is still very low.

CHAPTER 8

Conclusion

The goal of this thesis was to provide persistence for the space based middleware MozartSpaces, an implementation of the XVSM specification. Previous attempts to do this had some drawbacks and a new approach had to be found.

This goal was achieved by implementing a new persistence layer that replaces the Java collections where data was stored in previous versions of MozartSpaces. The persistence layer is integrated into the core of MozartSpaces and its transaction infrastructure in order to ensure compliance with the XVSM specification.

The persistence is configurable and it offers several persistence profiles. The in-memory profile emulates the transient nature of the previous MozartSpaces version with only a small performance overhead caused by the new abstraction layer – CAPI-3 operations became ca. 10–30% slower. Three profiles are based on Berkeley DB and are intended for scenarios which require fast persistence (at the cost of potential data loss in case of a crash) and those that require extreme data safety (at the cost of performance) as well as a reasonable compromise in between. Additionally the Android version of MozartSpaces can leverage the functionality of the SQLite database which is part of the operating system. Its performance is comparable to the Berkeley DB. The persistence is extensible and it is possible to add support for further database systems.

A benchmark suite was developed to evaluate the performance of the persistence. The results have shown that depending on the selected profile the persistence can be very fast and the performance overhead is an acceptable price for not losing all data when the execution terminates. The performance can be further optimized with configurable caches for various parts of the system. Storing the data of the space in a database also eliminated the need to constantly hold it in memory which has significantly reduced the memory consumption of MozartSpaces.

The goal of providing an energy-efficient persistence for the Android version of XVSM was reached only partially. This was caused by unsatisfactory tool support for

evaluation and optimization of energy-efficiency. Not surprisingly the benchmarks have shown that persistence increased power drawn by the CPU but unfortunately other hardware components like RAM and flash storage could not be measured. In general effect of the persistence on the energy efficiency was minimal when compared to to other hardware components such as the screen and wireless networks.

Bibliography

[AS92] B. Anderson and D. Shasha. Persistent linda: Linda+ transactions+ query processing. *Reasearch Directions in High-Level Parallel Programming Languages*, pages 93–109, 1992.

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

[Cat11] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39:12–27, May 2011.

[CH10] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[CKS09] Stefan Craß, Eva Kühn, and Gernot Salzer. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *IDEAS '09: Proceedings of the 2009 International Database Engineering & Applications Symposium*, pages 301–306, New York, NY, USA, 2009. ACM.

[Cod70] Edgar Frank Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13:377–387, June 1970.

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

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

[FHA99] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley Professional, 1999.

[FT05] Andreas Fongen and Simon J. E. Taylor. MobiSpace: A Distributed Tuplespace for J2ME Environments. In *17th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 202–207, 2005.

[Gar07] Matthew Garrett. Powering down. *Queue*, 5(7):16–21, November 2007.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7:80–112, January 1985.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings of the seventh international conference on Very Large Data Bases*, volume 7 of *VLDB '81*, 1981.

[ISO92] ISO. *ISO/IEC 9075:1992, Database Language SQL*. International Organization for Standardization, 1992.

[KAU12] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage for Smartphones. In *The 10th USENIX Conference on File and Storage Technologies*, FAST2012, February 2012.

[KK12] Je-Min Kim and Jin-Soo Kim. Androbench: Benchmarking the storage performance of android-based mobile devices. In Sabo Sambath and Egui Zhu, editors, *Frontiers in Computer Education*, volume 133 of *Advances in Intelligent and Soft Computing*, pages 667–674. Springer Berlin / Heidelberg, 2012.

[KL10] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51 –56, April 2010.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[KMS08] Eva Kühn, Richard Mordinyi, and Christian Schreiber. An Extensible Space-based Coordination Approach for Modeling Complex Patterns in Large Systems. *3rd Int. Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Special Track on Formal Methods for Analysing and Verifying Very Large Systems (ISoLA 2008)*, 2008.

[LCKW] Seongjin Lee, Seokhui Cho, Haesung Kim, and Youjip Won. Performance analysis of SSD/HDD hybrid storage manager. In *Nano, Information Technology and Reliability (NASNIT), 2011 15th North-East Asia Symposium on*.

[LCX+01] Tobin J. Lehman, Alex Cozzi, Yuhong Xiong, Jonathan Gottschalk, Venu Vasudevan, Sean Landis, Pace Davis, Bruce Khavar, and Paul Bowman. Hitting the distributed computing sweet spot with TSpaces. *Computer Networks*, 35:457–472, March 2001.

[Lib03] Leonid Libkin. Expressive power of SQL. *Theoretical Computer Science*, 296(3):379 – 404, 2003.

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

[Mar10] Alexander Marek. Design and Implementation of TinySpaces - The .NET Micro Framework based Implementation of XVSM for Embedded Systems. Master's thesis, Vienna University of Technology, 2010.

[Mei11] Thomas Meindl. XVSM Persistence - Developing an orthogonal functional profile for the eXtensible Virtual Shared Memory. Master's thesis, Vienna University of Technology, 2011.

[Mor10] Richard Mordinyi. *Managing Complex and Dynamic Software Systems with Space-Based Computing*. PhD thesis, Vienna University of Technology, 2010.

[NK07] Suman Nath and Aman Kansal. Flashdb: dynamic self-tuning database for nand flash. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 410–419. ACM, 2007.

[NVP10] Elena Nardini, Mirko Viroli, and Emanuele Panzavolta. Coordination in open and dynamic environments with tucson semantic tuple centres. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2037–2044, New York, NY, USA, 2010. ACM.

[O'N08] Elizabeth J. O'Neil. Object/Relational Mapping 2008: Hibernate and the Entity Data Model (EDM). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1351–1356, 2008.

[Pen10] Kostas Pentikousis. In Search of Energy-Efficient Mobile Networking. *Communications Magazine, IEEE*, 48(1):95 –103, January 2010.

[RC10] Sanam Shahla Rizvi and Tae-Sun Chung. Flash SSD vs HDD: High Performance Oriented Modern Embedded and Multimedia Storage Systems. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 7, pages 297–299, April 2010.

[RD05] Tomasz Rybicki and Jarosław Domaszewicz. MobileSpaces - JavaSpaces for Mobile Devices. In *Computer as a Tool, 2005. EUROCON 2005. The International Conference on*, volume 2, pages 1076–1079, November 2005.

[SBCD09] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Cáceres, and Nigel Davies. The Case for VM-Based Cloudlets in Mobile Computing. *Pervasive Computing, IEEE*, 8(4):14–23, October-December 2009.

[SHH10] Daniel Schall, Volker Hudlet, and Theo Härder. Enhancing energy efficiency of database applications using ssds. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, C3S2E '10, pages 1–9, New York, NY, USA, 2010. ACM.

[SLLP+10] Jacob Strauss, Chris Lesniewski-Laas, Justin Mazzola Paluska, Bryan Ford, Robert Morris, and Frans Kaashoek. Device transparency: a new model for mobile storage. *SIGOPS Oper. Syst. Rev.*, 44(1):5–9, March 2010.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[STG10] Weiqi Song, Tao Tao, and Tiegang Gao. Performance optimization for flash memory database in mobile embedded system. In *Education Technology and Computer Science (ETCS), 2010 Second International Workshop on*, volume 3, pages 35–39, March 2010.

[WCC04] George Clifford Wells, Alan G. Chalmers, and Peter G. Clayton. Linda implementations in Java for concurrent systems. *Concurrency and Computation: Practice and Experience*, 16(10):1005–1022, 2004.

[ZTQ+10] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM.

Web references

[1] Amazon. Amazon Simple Storage Service. http://aws.amazon.com/s3/. Accessed: 2012-03-26.

[2] Tim Bray. Saving Data Safely. http://android-developers.blogspot.com/2010/12/saving-data-safely.html. Accessed: 2012-03-27.

[3] Ed Burnette. Java vs. Android APIs. http://www.zdnet.com/blog/burnette/java-vs-android-apis/504, January 2008. Accessed: 2011-10-12.

[4] Oracle Corporation. Java SE 7 Features and Enhancements. http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html. Accessed: 2012-04-30.

[5] Couchbase. Couchbase. http://www.couchbase.com/. Accessed: 2012-02-01.

[6] Dan Creswell and other project contributors. The Blitz Project. http://www.dancres.org/blitz/. Accessed: 2012-03-16.

[7] H2 Database. H2 Database. http://www.h2database.com/html/main.html. Accessed: 2012-02-01.

[8] Stephen Deering and Robert Hinden. Internet Protocol, Version 6 (IPv6) Specification. http://www.ietf.org/rfc/rfc2460.txt, 1998. Accessed: 2012-03-26.

[9] Dropbox. Dropbox Anywhere. https://www.dropbox.com/anywhere. Accessed: 2012-03-26.

[10] Apache Software Foundation and its contributors. Apache River. http://river.apache.org/. Accessed: 2012-03-16.

[11] Google. Traceview. http://developer.android.com/guide/developing/debugging/debugging-tracing.html. Accessed: 2012-02-13.

[12] Google and guava project contributors. guava-libraries. http://code.google.com/p/guava-libraries/. Accessed: 2012-02-15.

[13] Android Developers (Google). Android platform version statistics. http://developer.android.com/resources/dashboard/platform-versions.html, September 2011. Accessed: 2011-09-25.

[14] Space Based Computing Group. SBC-Group. http://www.complang.tuwien.ac.at/eva/SBC-Group/sbcGroupIndex.html. Accessed: 2012-01-31.

[15] Space Based Computing Group. Mozartspaces. http://www.mozartspaces.org, 2011. Accessed: 2011-10-14.

[16] The hsql Development Group. HyperSQL. http://hsqldb.org/. Accessed: 2012-02-01.

[17] Qualcomm Incorporated. Trepn™Profile. https://developer.qualcomm.com/develop/development-devices/trepn-profiler. Accessed: 2012-02-12.

[18] Je-Min Kim. Androbench. http://www.androbench.org/. Accessed: 2012-03-28.

[19] eva Kühn, Geri Joskowicz, and Ralf Westphal. Xcoordination Application Space. http://xcoordination.org/. Accessed: 2012-04-24.

[20] Qusay H. Mamoud. Getting Started With JavaSpaces Technology: Beyond Conventional Distributed Programming Paradigms. http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/, July 2005. Accessed: 2012-03-16.

[21] University of Bologna. http://alice.unibo.it/xwiki/bin/view/TuCSoN/, 2011. Accessed: 2012-04-26.

[22] Oracle. Berkeley DB Java Edition documentation. http://docs.oracle.com/cd/E17277_02/html/java/com/sleepycat/je/Durability.SyncPolicy.html. Accessed: 2012-02-01.

[23] Oracle. JDBC Documentation. http://www.oracle.com/technetwork/java/javase/jdbc/index.html. Accessed: 2012-03-01.

[24] Oracle. VisualVM. http://visualvm.java.net/. Accessed: 2012-02-13.

[25] Oracle. WeakReference. http://docs.oracle.com/javase/6/docs/api/java/lang/ref/WeakReference.html. Accessed: 2012-02-03.

[26] Oracle. Oracle Berkeley DB Java Edition High Availability. http://www.oracle.com/technetwork/database/berkeleydb/berkeleydb-je-ha-whitepaper-132079.pdf, March 2010. Accessed: 2012-03-26.

[27] Checkstyle project contributors. Checkstyle. http://checkstyle.sourceforge.net/. Accessed: 2012-02-20.

[28] IBM Almaden Research. TSpaces FAQ. http://ftp.almaden.ibm.com/cs/TSpaces/faq.html#general. Accessed: 2011-10-16.

[29] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard), March 2011. Accessed: 2012-03-26.

[30] SQLite. Datatypes In SQLite Version 3. http://www.sqlite.org/datatype3.html. Accessed: 2012-02-17.

[31] SQLite. SQLite documentation. http://www.sqlite.org/omitted.html. Accessed: 2012-02-01.

[32] SQLite. SQLite Query Language: CREATE TABLE. http://www.sqlite.org/lang_createtable.html. Accessed: 2012-02-20.

[33] GigaSpaces Technologies. GigaSpaces. http://www.gigaspaces.com/. Accessed: 2012-04-29.

[34] University of Michigan. PowerTutor. http://powertutor.org/. Accessed: 2012-02-12.

[35] Stefan Weinbrenner and Adam Giemza. SQLSpaces. http://sqlspaces.collide.info/.

APPENDIX A

Performance benchmarks

This appendix describes the benchmarks that were used to measure the performance of the persistence layer in MozartSpaces.

The number of entries used in the benchmarks are for the J2SE version. During the execution of the benchmarks on the Android operating system the number of entries was 200 times smaller to compensate for the significantly slower hardware in order to make the execution time more practical.

Unless stated otherwise the benchmarks always use a single short `String` as entry value.

Write benchmarks always start with an empty container with the AnyCoordinator.

- **Write - multiple TX:** each thread writes $100000/threads$ entries in a batch of 50 entries per transaction.
- **Write - single TX:** each thread writes $100000/threads$ entries in a single transaction.
- **Write - separate TX:** each thread writes $100000/threads$ entries each in its separate transaction.

Read benchmarks prefill a container with an AnyCoordinator with 2000 entries.

- **Read all:** each thread retrieves all entries $1000/threads$ times.
- **Read - separate Tx** each executes $50000/treads$ read operations each in a separate transaction.

Take benchmarks prefill a container with AnyCoordinator with 10000 entries.

- **Take all:** each thread tries to take as many entries as possible.
- **Take many - single Tx:** each thread takes $10000/threads$ entries.
- **Take - single Tx:** each thread takes one entry at a time until the container is empty.
- **Take - separate Tx:** like previous benchmark but each take operation has its own transaction.

FifoBenchmarks evaluate the performance of a container with the FifoCoordinator.

- **Fifo add - single Tx:** each thread writes $10000/threads$ entries.
- **Fifo read all:** each thread reads all entries from a container with 100000 entries.
- **Fifo read - single Tx:** each thread executes $100000/threads$ read operations from a container with 100000 entries.
- **Fifo take - single Tx:** each thread executes $10000/threads$ take operations from a container with 100000 entries.
- **Fifo take many - single Tx:** each thread takes $20000/threads$ entries from a container with 100000 entries.

KeyBenchmarks evaluate the performance of a container with the KeyCoordinator.

- **Key add - single Tx:** each thread writes $10000/threads$ entries.
- **Key read - single Tx:** each thread executes $100000/threads$ read operations from a container with 100000 entries.
- **Key take - single Tx:** each thread executes $100000/threads$ take operations from a container with 100000 entries.

VectorBenchmarks evaluate the performance of a container with the VectorCoordinator.

- **Vector add - single Tx:** initially the vector is prefilled with two entries and then $2000/threads$ entries are added always at the second position always shifting all subsequent entries in the vector to the right.

- **Vector append - single Tx:** append $1000/threads$ entries to an initially empty vector.
- **Vector read - single Tx:** read $10000/threads$ entries from a container prefilled with 10000 entries.
- **Vector take - single Tx:** take $10000/threads$ entries from a container prefilled with 10000 entries.

LindaBenchmarks evaluate the performance of a container with the LindaCoordinator. Unlike the previous benchmarks the LindaCoordinator also has to analyze the actual entry value. In this case the entries are simple POJOs with that contain one `String` and one `long` which is used for template matching.

- **Linda add - single Tx:** each thread writes $1000/threads$ entries.
- **Linda read - single Tx:** each thread executes $10000/threads$ read operations from a container with 10000 entries.
- **Linda take - single Tx:** each thread executes $10000/threads$ take operations from a container with 10000 entries.