FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Design and Implementation of the next Generation XVSM Framework

Runtime, Protocol and API

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Tobias Dönz
Matrikelnummer 0226173

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuerin: A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn

Wien, 06.06.2011 ____________________ ____________________

(Unterschrift Verfasser) (Unterschrift Betreuerin)

Tobias Dönz, Neustiftgasse 83/217, 1070 Wien

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

Wien, 06.06.2011

____

(Unterschrift)

Abstract

The development of software for distributed systems is often carried out with middleware that provides programming abstractions to ease the implementation. Space-based middleware systems are based on tuple spaces and allow loosely coupled components of a distributed system to coordinate over a shared data store. They provide synchronization, time and space decoupling. However, tuple spaces offer no order for the stored data objects which limits the coordination capabilities. eXtensible Virtual Shared Memory (XVSM) is a space-based middleware architecture that allows for more and flexible coordination and can be easily extended. Recently, it has been specified with a formal model in several layers, of which the lower ones have been implemented in Java as previous work.

This thesis describes the design and implementation of the upper layers of the formal model in MozartSpaces 2.0, the Java implementation of XVSM. They comprise the runtime, remote communication and the API. The runtime supports blocking operations with timeouts and aspects to extend the middleware functionality. Important is the fast scheduling of requests and the efficient handling of internal events for blocking operations. As example for the extensibility of XVSM we implement publish/subscribe notifications with aspects. We also present an XML protocol for the interoperable remote access to XVSM spaces and a synchronous and asynchronous API. Benchmark results show that MozartSpaces 2.0 is considerably faster than the previous version 1.0 for all operations and scales better.

Kurzfassung

Zur Entwicklung von Software für verteilte Systeme wird häufig Middleware verwendet die Programmierabstraktionen bereitstellt und damit die Implementierung vereinfacht. Space-basierte Middleware-Systeme basieren auf Tuple Spaces und ermöglichen lose gekoppelten Komponenten eines verteilten Systems die Koordination über einen gemeinsamen Datenspeicher. Sie bieten Entkopplung bezüglich Synchronisierung, Zeit und Raum. Allerdings haben die in Tuple Spaces gespeicherten Datenobjekte keine Ordnung, wodurch die Koordinationsmöglichkeiten eingeschränkt sind. eXtensible Virtual Shared Memory (XVSM) ist eine space-basierte Middleware-Architektur welche mehr und flexible Koordination ermöglicht und einfach erweitert werden kann. Kürzlich wurde sie mit einem formalen Modell in mehreren Schichten spezifiziert. Die unteren Schichten wurden im Rahmen einer anderen Diplomarbeit in Java implementiert.

Diese Diplomarbeit beschreibt das Design und die Implementierung der oberen Schichten des formalen Modells in MozartSpaces 2.0, der Java-Implementierung von XVSM. Diese umfassen die Runtime, die Remote-Kommunikation und die API. Die Runtime unterstützt blockierende Operationen mit Timeouts und Aspekte, mit denen die Middleware-Funktionalität erweitert werden kann. Wichtig ist dabei das schnelle Scheduling von Requests und das effiziente Verarbeiten von internen Events für die blockierenden Operationen. Als Beispiel für die Erweiterbarkeit von XVSM implementieren wir "Publish/Subscribe"-Benachrichtigungen mit Aspekten. Wir präsentieren auch ein XML-Protokoll für den interoperablen Remote-Zugriff auf XVSM-Spaces und eine synchrone und asynchrone API. Benchmark-Ergebnisse zeigen, dass MozartSpaces 2.0 für alle Operationen erheblich schneller als die Vorgängerversion 1.0 ist und besser skaliert.

Danksagung

Diese Diplomarbeit steht am Ende meines Studiums und wäre ohne die Unterstützung durch einige Personen nicht möglich gewesen.

Zuerst möchte ich mich bei meinen Eltern bedanken, die mir das Studium ermöglicht und mich dabei unterstützt haben. Maßgeblich für den erfolgreichen Abschluss meines Studiums ist auch eva Kühn, die mir die Möglichkeit gab diese Arbeit zu erstellen, und mich dabei fortlaufend unterstützt hat. Bei ihr und Stefan Craß möchte ich mich besonders für die konstruktiven Anregungen und Korrekturen sowie ihre Geduld bedanken.

Darüber hinaus möchte ich mich auch bei den weiteren Mitarbeitern der Space Based Computing Group und den Mitgliedern des XVSM Technical Board, im Besonderen Martin Barisits und Richard Mordinyi, für die anregenden Diskussionen und ihr Feedback bedanken.

Diese Diplomarbeit wurde durch das folgende Forschungsprojekt unterstützt: FIT-IT (Forschung, Innovation und Technologie für Informationstechnologien) project 825750, Secure Space – A Secure Space for Collaborative Security Services.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API	Application Programming Interface
CAPI	Core API
FIFO	First-In First-Out
GC	Garbage Collector
JVM	Java Virtual Machine
MOM	Message-Oriented Middleware
OOME	Out Of Memory Error (`java.lang.OutOfMemoryError`)
SD	Standard Deviation
SXQ	Simple XQ
XP	XVSM core Processor
XQ	XVSM Query
XVSM	eXtensible Virtual Shared Memory
XVSMP	XVSM Protocol
XVSMQL	XVSM Query Language

1 Introduction

Today most computers are connected to a network and part of a distributed system like the internet. In general, a distributed system is formed by computers that are independent but look like a single coherent system to the user [TS07]. Components on this computers pass messages to communicate and coordinate their actions [Cou05]. Compared to components on one computer, the communication between computers is more complex, also because the computers in a distributed system can be heterogeneous, for example, they can run different operation systems. There *middleware* comes into play. Middleware is a software layer between the operating system with basic network communication abilities and application components. It is an abstraction that eases the development of distributed applications and may partly hide the fact that an application is distributed.

There are different types of middleware regarding the programming abstraction they offer. [Bak03] classifies middleware into four categories—distributed tuples, message-oriented, distributed object and remote procedure call (RPC) middleware. A similar classification can be found in [Emm00]. *RPC middleware* allows to call procedures over the network, similar to local procedure calls. *Distributed object middleware* extends the RPC functionality for objects of object-oriented programming languages. *Message-oriented middleware* (MOM) allows for message exchange, usually in form of message queues. *Distributed tuples* is used in [Bak03] as category for distributed relational databases with transactions (and transaction monitors) and tuple spaces, a form of a shared memory.

One important characteristics for a middleware class is that of coupling, that is, which forms of (de)coupling it supports. [AvdADtH05] analyzes the coupling dimensions *synchronization*, *time* and *space*. Synchronization (or thread) decoupling allows non-blocking communication and thereby the interleaving of processing (computation) and communication. Sending and receiving messages can be either blocking or non-blocking and blocking/non-blocking combinations are allowed (partial synchronization decoupling). Time decoupling does not require sender and receiver to be active at the same time, and therefore needs obviously another participant, a place or medium to (temporarily) store the messages. Space decoupling allows indirect and anonymous communication, sender and receiver do not need to know or hold references of each other. [AvdADtH05] also classifies middleware solutions and standards with the coupling dimensions. RPC and distributed object middleware provide less decoupling than MOM or tuple spaces, which both offer space, time and partial synchronization decoupling[1]. Therefore the use of MOM or tuple spaces generally leads to comparatively less coupling or loose coupling.

Loose coupling is often desired in distributed systems and can be easily achieved with middleware that allows for decoupling. An example is, when producers and consumers of data work at different or varying speeds. A message queue or tuple space could be used as buffer here. Another example is, when mobile devices that are not always online need to communicate with each other. Time decoupling is important in such a scenario and MOM or tuple spaces could be used. A (relational) database would also be an option in this case. But databases generally do not support blocking operation like most tuple spaces do as it is explained later.

MOM implementations like Java Message Service (JMS) [HBS$^+$02] allow processes to send and receive message over specific queues (point-to-point model) or publish and subscribe to messages categorized by a topic (publish/subscribe or pub/sub model). With the pub/sub model we get time, space and synchronization decoupling [EFGK03]. A tuple space is an abstraction for accessing a (virtual) shared memory and was introduced to distributed systems with the Linda coordination model [Gel85]. Tuples are collections of data values (like messages in MOM) and they can be stored to and retrieved from a tuple space by the components in the distributed systems. A tuple space implicitly provides time and space decoupling, as tuples are stored independent of receivers and are retrieved based on their content. With non-blocking operations to access the tuple space we also get (partial) synchronization decoupling.

While MOM and tuple spaces can offer the same degree of decoupling, they differ in other properties as a comparison in [Mar10b] shows. With tuple spaces, components communicate through a shared data store, the coordination is data-driven. With the original point-to-point model of MOM, components are connected with channels through which messages, that are controlled by the components, flow. The coordination is control-driven (see also [Mor10]). Both middleware concepts were extended and enhanced over time, so that recent implementations are more alike than the original models, but the communication direction and the order of data objects (messages or tuples) are still different. MOM facilitates unidirectional com-

[1][AvdADtH05] lists the tuple space middleware implementations JavaSpaces and several MOM middleware systems with *partial* synchronization decoupling because the sending of messages blocks and only the receiving is non-blocking. We believe this is only an implementation detail/issue. MozartSpaces 2.0 offers full synchronization decoupling.

munication with a fixed order of the messages, generally first-in first-out (FIFO). In contrast, tuple spaces facilitate undirected communication without a defined order of tuples (random access). Tuples are retrieved with templates that can match one or more, possibly all, tuples. Subsequent read operations on an unchanged tuple space may return different tuples. If a use case requires a specific order of tuples, this has to be introduced and managed manually with the aid of sequence numbers. With such additional effort tuple spaces can simulate MOM. The reverse is also true. JMS, for example, can simulate the content-based retrieval with Message Selectors for use cases where the FIFO order is not sufficient, although small differences remain (see [Mar10b] for details). Overall, the use of MOM or tuple spaces with a data item order that is different from the default order of the model is possible but cumbersome.

Extensible Virtual Shared Memory (XVSM) [11] is a middleware concept that is influenced by the Linda model and other tuple spaces. One of its core functionality is the support of different *coordinators* that can support arbitrary (complex) coordination patterns and provide, for example, Linda-like selection of data items or a FIFO order like MOM. XVSM has been described first in 2005 [KBM05] and was later implemented for the Java platform, named MozartSpaces [Sch08b, Prö08], and for the .NET platform as XcoSpaces [Sch08a, Kar09]. The two implementations were used in several projects for different use cases. The experience gained by these applications and the efforts for making the implementations interoperable showed where the XVSM model could be improved and that a more detailed formal description is required. Hence, a formal model of XVSM has been developed as basis for new or adapted implementations [CKS09, Cra10].

MozartSpaces 2.0

For the Java implementation of the new formal model, the new MozartSpaces version 2.0, it was decided to implement it completely from scratch, which is partly described in this thesis. As it will be explained in more detail in Chapter 3, XVSM has internally a layered architecture. The core of XVSM is divided into four core API (CAPI) layers. In MozartSpaces the layers CAPI-1 to CAPI-3 are combined, their architecture and implementation is described in [Bar10]. The CAPI-3 layer has an interface that provides methods to access data stored in the space with transactions and above-mentioned coordinators.

The objective of this thesis is to realize the CAPI-4 layer, the MozartSpaces *runtime* which supports blocking operations with timeouts and *aspects* that can extend the functionality. It also treats the remote communication, the interoperable protocol of XVSM (XVSMP) and the low-level embedded user API. The core functionality of the runtime is the fast scheduling of incoming remote and embedded requests and the dispatching of responses. Compared with the previous version MozartSpaces 1.0, the new runtime should provide better performance and scalability. Special attention should be paid to the prevention of concurrency related race conditions and a clean, modular architecture that allows for easy extension. The user API should also allow the asynchronous (non-blocking) invocation of space operations, besides synchronous (blocking) calls. Of course the implementation should also conform to the new formal model.

Main Results

In this thesis we present the architecture and implementation of the runtime for MozartSpaces 2.0. It is implemented on top of the CAPI-3 implementation from [Bar10] and completes the first implementation of the formal model of XVSM in Java. To support blocking operations, the formal model describes the use of internal events and a *wait container* for blocked requests. We present and discuss an alternative structure to manage the events and waiting requests. XVSM provides transactions and uses locks on data structures to isolate them. We designed and implemented a graph-based module to detect deadlocks of transactions because of locks in MozartSpaces.

To allow for the interoperable remote communication with other platforms, we designed and implemented the XVSMP based on an XML schema definition. To access an XVSM space, the low-level user API that supports synchronous and asynchronous calls can be used.

Benchmark results show that MozartSpaces 2.0 is much faster than the previous version. This is the case for all operations with embedded and remote access. Concurrent tests show also that MozartSpaces 2.0 scales better than version 1.0. Altogether, the implementation fulfills the performance requirements.

Thesis Structure

This thesis is organized as follows: Chapter 2 describes tuple spaces, the original Linda model and tuple space middleware implementations for Java. They influenced the development of the space-based middleware model XVSM that is described in Chapter 3. The architecture of the MozartSpaces 2.0 runtime is explained in

Chapter 4. Chapter 5 continues with a description of the extensibility, the remote communication and the interoperable XML protocol of MozartSpaces. The low-level embedded user API is presented in Chapter 6. Chapter 7 contains a short description of the current structure of the MozartSpaces 2.0 implementation. The evaluation in Chapter 8 provides a performance evaluation of the internal processing in the MozartSpaces 2.0 runtime and benchmarks where MozartSpaces 2.0 is compared to MozartSpaces 1.0. Chapter 9 lists some concrete ideas for future work on XVSM and MozartSpaces, before we conclude in Chapter 10.

2 Tuple Spaces

This chapter gives an overview of important tuple space models and some implementations. Section 2.1 describes the seminal Linda model, where a tuple space is used as data store. Section 2.2 gives a brief overview of TSpaces, a Java tuple space implementation. JavaSpaces is a tuple space specification for Java with several implementations, described in Section 2.3. Section 2.4 points to comparisons of further tuple space implementations and describes some recent implementations which are not included in them.

2.1 Linda

Linda was developed in the 1980's at Yale University and was introduced by David Gelernter as basis for a distributed programming language in [Gel85]. It uses a *tuple space* (space) as basis for *generative communication.* Processes communicate over the space with tuples, a series of typed fields, which can be passive (data values/objects) or active (executable code). The communication is called *generative* because the (generated) tuples exist in the space independent of the communicating processes. In [GC92] Linda was termed a *coordination language* that forms a *complete programming language* when it is combined with a *computation language.*

The space is accessed with six operations to generate, read and consume tuples [CG89]. `out` generates a tuple and adds it to the space. `eval` creates an active tuple that is executed in a new process. The result is added as passive tuple to the space. The operation `rd` reads a tuple from the space by returning a copy of it. `in` takes a tuple from the space where it is removed (consuming read). `out` and `eval` are non-blocking, but `rd` and `in` block when no matching tuple is available. The non-blocking *predicate* variants for `in`, `inp`, and `rd`, `rdp`, return an error when no matching tuple is found in the space. Selecting a matching tuple is performed associatively with a template (*template matching*). A template is similar to a tuple, but for each field it can have a value (actual parameter) or a wildcard (formal parameter) where only the type but not a value is specified. A tuple in the space matches a template when the values of the tuple's fields are equal to the template's fields (actual parameter) or of the same type (formal parameter). Tuples in the space have no order and are matched and returned nondeterministically.

2.2 TSpaces

TSpaces [WMLF98, LCX$^+$01] is a tuple space middleware for Java developed by IBM. The project has been stopped but the software is still available as a free download [10]. In TSpaces classes for tuples and fields are provided. There are operations to write, read and take a tuple, similar to Linda but with other names and without the `eval` functionality. The blocking operations to read and take a tuple support a timeout, after which they return without a tuple. The TSpaces API provides also a method to delete a tuple and operations to write, read, take and delete multiple tuples as well as count how many tuples match a given template. Tuples have an ID that can be used to read, take or update (replace) a specific tuple and can have an expiration time, after which they are automatically removed from the space. TSpaces also supports transactions that span multiple operations. For event notifications, a callback method can be specified, which is called when a tuple matching a template is written to the space.

TSpaces supports template matching similar to Linda and subtype matching, that is, tuples also match if the types of tuples and their fields are subtypes of the types of the template and its fields. Furthermore, it offers several types of queries as alternative to template matching. Indexed queries match fields as name-value pairs with optional value ranges. Tuples can use special XML fields which can be queried with a subset of the early XML query language XQL. Queries can also be combined with *AND* and *OR* operators. An additional space operation, `rhonda`, allows two processes to atomically synchronize and exchange data over the space with template matching. Moreover, command handlers can be added to a space, custom functions that can be addressed with a name and extend the space functionality. In TSpaces, a server can run standalone or within the application that accesses it. A server supports multiple (named) tuple spaces per instance and optionally persistence, that is, the tuple spaces and their content will be saved across server restarts.

2.3 JavaSpaces

JavaSpaces [FHA99] is a tuple space specification for Java. The tuples are called entries and are special Java objects that can be exchanged and coordinated with a JavaSpaces service. Such a service provides an interface `JavaSpace` with several methods to access the space. The basic operations are `write` to insert an entry, `read` to get an entry matching a given template and `take` to get and remove a matching entry. JavaSpaces specifies template matching similar to the Linda model, with the extension of subtype matching like in TSpaces. There are blocking and non-blocking operations to read and take entries, both with a timeout parameter. The functionality of the Linda `eval` is not provided by JavaSpaces. Event notifications are supported with `notify`, which registers a callback method that is called when an entry matching a template has been written to the space. The interface `JavaSpace05` was added later and extends the interface `JavaSpace` with methods to write, read and take multiple entries as well as register for special events in combination with transaction visibility. Distributed transactions and leases, a way to manage an expire time and the expiration of resources like entries or transactions, are provided by Jini. In fact, JavaSpaces is part of the Jini technology, an architecture and environment for distributed systems [Wal00].

Jini is based on Java Remote Method Invocation (RMI) and defines infrastructure services like discovery, distributed transactions, leases, events and so on. The Jini Technology Starter Kit provides an implementation of these services and contains the JavaSpaces implementation Outrigger. Originally developed by Sun Microsystems, it is now continued as the Apache River project [20]. Other currently actively developed JavaSpaces implementations are Blitz [3], which is open source, and GigaSpaces [7], which is commercial software that provides many extensions and additional services in addition to the JavaSpaces specification. Outrigger, Blitz and GigaSpaces support transient and persistent spaces. A JavaSpaces implementation based on MozartSpaces 1.0 has been implemented and is described and evaluated in [Kes08].

2.4 Other Models and Implementations

There exist many models and implementations for tuple spaces besides the ones described above. Several of them were illustrated and compared to XVSM in earlier publications. [Kes08] describes the JavaSpaces implementations Outrigger, Blitz and GigaSpaces and the concepts of the tuple spaces ActiveSpace, Blossom, LIME, LighTS, MARS, TSpaces, TuCSoN, XMLSpaces and XMLSpaces.NET, as well as the space-based middleware Corso. [Sch08a] classifies and compares the middleware solutions Blitz, GigaSpaces, LightTS and Corso that are based on a shared data space model. The classification includes the coordination concepts, the supported operations, the extensibility and the architecture. [Mor10] briefly describes several coordination frameworks that are related to the Linda model and gives an overview of their coordination and querying capabilities and which data format(s) they support. His comparison includes the frameworks ActiveSpace, ATSpace, B-Linda, Bauhaus Linda, Blossom, BONITA, Corso, DTuples, eLinda, GigaSpaces, GLinda, Grinda, JADA, JavaSpaces, Kernel Linda, KLAIM, LIME, LighTS, LuCe, MARS, P4-Linda, PLinda, PoliS, SwarmLinda, TSpaces, TuCSoN and XMLSpaces. Independent of XVSM, a comparison of the original Linda model (Yale Linda) to the Java implementations TSpaces, JavaSpaces and GigaSpaces can be found in [WCC04], along with a description of the author's system eLinda.

By searching for current space-based middleware, we found several new tuple space implementations that are not mentioned in the references above. Four of them are described below. For TIBCO ActiveSpaces [23] (not to be confused with an older tuple space implementation named *ActiveSpace*, which is included in the review of [Kes08]), not much information was found. It is a proprietary, commercial product that cannot be downloaded (for free), and only a short description and superficial documentation is available online.

2.4.1 SemiSpace

SemiSpace is an open source tuple space implementation for Java [12]. It was inspired by JavaSpaces, but it neither implements the JavaSpaces specification nor uses Jini. Every Java object can be used as an entry, if it can be serialized by XStream [25], the XML serialization library which is used internally to handle the data objects. The API provides operations to write, read and take entries. The blocking read and take operations also have a timeout parameter. Template matching is supported, but without subtype matching. Event notifications can be registered similar to JavaSpaces. Furthermore, leases are used for entries and event registrations, that is, they expire after a specified duration, if the lease is not renewed. In contrast to JavaSpaces, SemiSpace does not support transactions.

SemiSpace includes several optional modules that provide additional features. It can be clustered with Terracotta [18] or combined with the integration framework Apache Camel [21]. Furthermore, a SOAP web service interface with a Java client API exists, and a JavaScript API that can also be used from web browsers.

2.4.2 SQLSpaces

SQLSpaces is another open source tuple space for Java [2]. It provides an interface that is very similar to TSpaces, and the research group that develops it used TSpaces in their earlier work [WGH07]. The basic functionality for tuple handling, event notifications and transactions is practically the same as in TSpaces (see Section 2.2).

As extensions, read and take operations can also be performed over multiple spaces of a server with one call. A component called Object Tuple Mapper (OTM) allows to write arbitrary data objects, whose classes are specifically annotated, to the space, without the need to implement the SQLSpaces types for tuples and their fields. The entry storage is performed with a relational database. Several database systems are supported, the transactional semantics and the persistence behavior depends on the one that is used. Space calls are internally handled as XML strings. They can also be performed over a SOAP web service interface. SQLSpaces also supports versioning for spaces. API methods are provided to create snapshots of one or more spaces and access a specific snapshot of a space. While SQLSpaces is implemented in Java, there are clients with limited functionality for other languages, currently C#, Ruby, Prolog and PHP. Furthermore, a browser-based user interface allows to investigate the contents of spaces and issue commands through a command-line interpreter (shell).

2.4.3 Fly Object Space

Fly Object Space (Fly) is a tuple space with native implementations for several operating systems (Windows, Linux, Mac OS X and Solaris). It is not open source but freely available for non-commercial use [27]. A Fly server can be accessed with one of the language bindings (clients) for Java, Scala and Ruby. They use a binary protocol called Fly Binary Interface to access the server. The Java client has operations to write, read and take one or many entries. Arbitrary objects can be used as entries, but the documentation does not mention interoperability between different platforms, e.g, between Java and Ruby. Event notifications are supported by registering a callback method. Leases are used for entries and notifications. Transactions are not supported. Fly supports lookup in a local network with multicast. It stores entries only transiently in-memory.

2.4.4 Gruple

Gruple is an open source tuple space implementation written in Groovy, a programming language for the Java Virtual Machine (JVM), like Java [26]. It has currently no support for remote access and can thus only be used for in-process coordination. Gruple supports multiple spaces and the interface to access a space provides methods to write (`put`), read (`get`), and take (`take`) tuples. A tuple is a map and can contain immutable standard or user-defined types. Template matching is used for the selection of tuples. As extension to values and wildcards, closures can be used in templates. Gruple provides transactions that can also be used to access several spaces. The spaces are stored only in-memory and are not persisted.

2.4.5 Conclusion

The comparisons of space-based coordination frameworks in [Mor10] and [Sch08a] show that XVSM has unique features like the flexible coordination and the extensibility with aspects. The implementations described above do not change that situation. SemiSpace has extensions for clustering and integration, but cannot be extended itself. SQLSpaces provides interesting features like versioning or coordination capabilities that extend the Linda template matching, but they are not as flexible and extensible as the XVSM custom coordinators.

3 Extensible Virtual Shared Memory

In this chapter we give an overview of the space-based middleware architecture XVSM. In Section 3.1 we present the basic concepts of XVSM. Section 3.2 describes the development of XVSM, the already existing implementations and their applications. The new formal model of the XVSM core is outlined in Section 3.3 and the concept of profiles to extend the core functionality of XVSM is presented briefly in Section 3.4.

3.1 Basic Concepts

At the heart of XVSM is the space. The data objects stored in an XVSM space are called *entries*. They are not directly stored in the space but in *containers* that structure a space and can be seen as subspaces. The entries in a container are managed by one or more *coordinators*. There are different coordinator types, for example, a *FIFO* coordinator that ensures FIFO order of entries, or a *Linda* coordinator that supports Linda template matching on entries. Figure 3.1 shows an example of a container with three coordinators and eight entries. In this example all entries are managed by the coordinator *Random*, an *obligatory coordinator*, but only a subset by the coordinators *FIFO* and *Key*, which are used as so called *optional coordinators*. XVSM also supports *transactions* to perform several of the subsequently explained operations atomically, either all or none of them, and isolate concurrently running transactions from each other.

XVSM has operations to create and destroy a container. Furthermore, a container can have a name or it can be unnamed. If it has a name, it can be looked up with the according operation. Additionally, a container can be exclusively locked for a transaction. Further operations are provided to write entries to a container and read, take or delete them from a container. *Coordination data* can be provided for an entry that is written to the container. It is additional coordinator-specific data like the key for an entry that is written to a container with a Key coordinator (cf. Figure 3.1). When a read, take or delete operation is called, a *selector* must be specified to select the entries on which the operation is performed. Thus

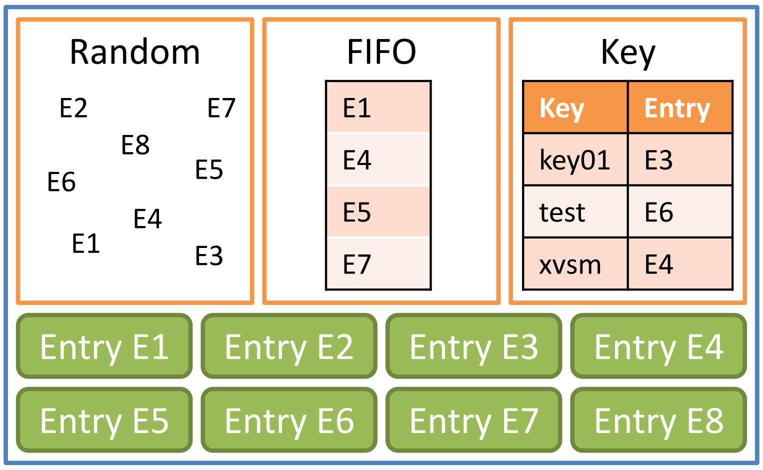


Figure 3.1: Container with several coordinators and entries

these operations are called *selecting operations*. A selector is coordinator-specific, for example, a FIFO selector is necessary to select entries managed by a FIFO coordinator. The difference between the selecting operations is that read returns the selected entries, take removes them from the container and returns them, and delete just removes them from the container. The selecting operations in general allow to select multiple entries, but the selection and thus also the number of entries that are returned is controlled by the specific selectors and coordinators. Moreover, the selecting operations can be called with a timeout that controls the blocking behavior, that is, whether and how long the operation should block. Multiple entries and a timeout is also supported for the write operation. In contrast to other tuple spaces, in XVSM this operation can also block, because a container can be bounded, that is, it can have a maximum number of entries. All container and entry operations support transactions. For the transaction control, operations to create and commit or rollback a transaction are provided. A transaction has a timeout, after which it is automatically rolled back, if it has not been committed or rolled back yet.

The description above is for a single space. While XVSM can be used to coordinate threads within one process, it is even more suitable for the collaboration of processes on different computers (peers) in a distributed system. Applications on different peers can access their own (embedded) space or other (remote) spaces with the same API. Figure 3.2 shows the different variants of an XVSM instance or core. *Core A* is an instance with a space that the user application in the same process can access directly without remote communication. The core API can be used to access the embedded space and any remote space; the remote communication is hidden. *Core B* is a server instance of a space that is accessed only remotely, there is no user application that could use the core API. *Core C* is just a client without an own space. The difference between the access to the embedded container *C2* and the remote container *C5* by *Core A* in Figure 3.2 is only the use of a different name to refer to the location of the container.

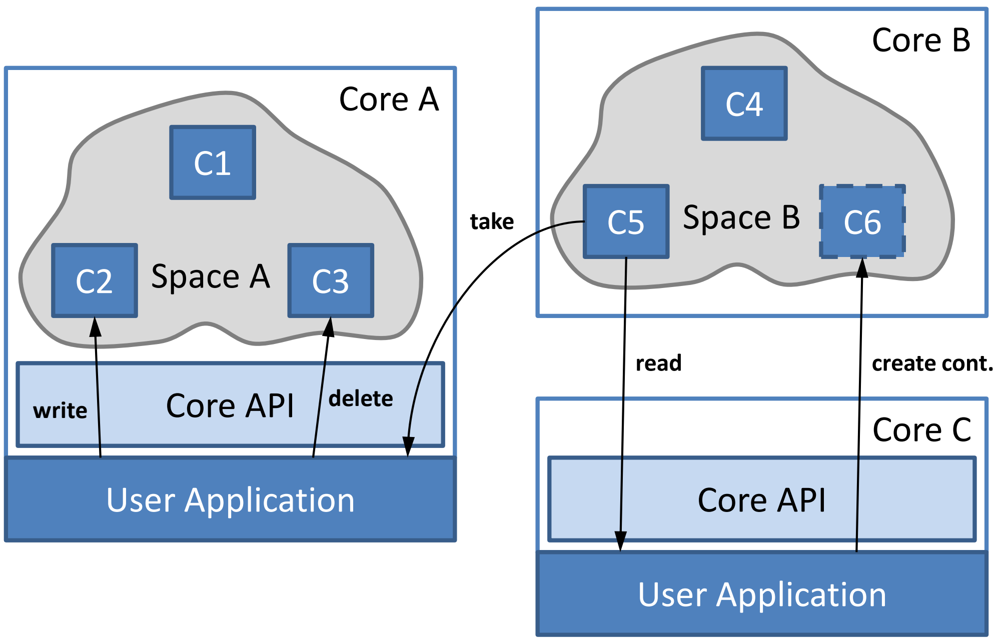


Figure 3.2: XVSM peers with embedded and remote space access examples

We have now described different peers with a space (or just a client API) that can coordinate each other over one or more spaces. This is what the XVSM core provides. An extension on top of the core could hide the different spaces on the peers and provide the user with the view of a common space, where a single space location is not directly addressed. Such an extension is an example for a so-called *profile*, a concept to extend XVSM that is described in Section 3.4.

3.2 Existing Implementations

XVSM is mainly influenced by the Linda model, tuple spaces that evolved from Linda, and Corso [Küh94], a virtual shared memory model also developed at the Institute of Computer Languages of the TU Vienna. First concepts of XVSM were published in 2005 and prototyped with Corso [KBM05]. Afterwards XVSM was implemented independently of Corso and in 2006 the first Java implementation MozartSpaces was created. Version 1.0 was published in 2008 and is described in [Sch08b] and [Prö08]. Later, and partly in parallel, a .NET implementation of XVSM was created and named XcoSpaces. It is described in [Sch08a] and [Kar09]. These two implementations can interoperate via an XML protocol, with some limitations regarding the data types of entries. They also support most of the basic functionality of XVSM described in Section 3.1. However, the semantics of the operations is not clearly defined and no formal model exists. Thus, the implementations do not behave exactly the same and some workarounds are required for interoperability. Because of these issues, ideas for improvements, experiences from building applications with XVSM and the goal to eventually formally verify XVSM, a formal model was developed. An early version of this model was published in [KMS08b], parts of the current version in [CKS09]. The most complete description of the formal model and a Haskell prototype based on it is presented in [Cra10]. While the formal model was developed, an XVSM implementation for embedded devices was created [Mar10a]. It is termed TinySpaces, uses the .NET Micro Framework and partly follows already the formal model, though with limitations because of the restricted environment of embedded devices.

3.2.1 Applications

Previous implementations of XVSM were used in a university course about space-based middleware and several research projects with industry. For example, it was used in an application for Intelligent Transport Systems (ITS) together with a peer-to-peer (P2P) framework and some extensions [BFK+11, KMG+09]. The P2P framework has been integrated into MozartSpaces 1.0 as profile with aspects (see Section 3.4) and was used for the distributed lookup and replication of space containers with volatile geo-tagged data. The content-based access with Linda template matching was used to retrieve the data. Another example is the Self-Initiative Load Balancing Agents (SILBA) pattern where load balancing agents coordinate themselves with a shared space [KSC09, SC11]. Because of the undirected and loosely-coupled communication the space provides, it is well suitable for this use case. The prototype used to implement the pattern uses also several different XVSM coordinators. Further projects and applications are described by Richard Mordinyi in [Mor10], where XVSM is also used as integration platform and as an abstraction layer for agile software development. Because of its flexible coordination capabilities, it can directly support different coordination models and architectural styles (see also [MKS10]).

Although XVSM implementations were used successfully in several applications, performance measurements show mixed results. A comparison of the two XVSM implementations MozartSpaces 1.0 and XcoSpaces with the JavaSpaces implementations GigaSpaces and Blitz can be found in [KMM+08]. The benchmark in embedded mode shows that GigaSpaces is fastest, followed XcoSpaces, Blitz and then

MozartSpaces. Interestingly XcoSpaces is much faster than MozartSpaces. Another benchmark in [Löw08] also shows that MozartSpaces 1.0 is relatively slow. This was one of the reasons for creating a new implementation from scratch.

3.2.2 Runtime Architecture

The runtime model of the previous XVSM implementations is depicted in Figure 3.3 and consists of components that process messages, e.g., space operation requests, and containers to queue these messages for the components. This model follows the staged event-driven architecture (SEDA), where event-driven stages are connected and decoupled by event queues [WCB01]. SEDA stages have internally an event handler, that runs on worker threads of a thread pool and processes incoming events from the event queue. A resource controller manages the thread pool and queue.

In the XVSM runtime model an event (or message) is either a *request*, a *response* to a request, or an internally used XVSM *event* for handling blocking operations. The *core processor* (XP) takes requests from the *request container*, executes them, and handles the results. If the result is OK, a response is written to the *response container* and an event is written into the *event container*, if a waiting request could be woken up by it. In case of error an according response is written to the response container. If there are not enough or too many entries to fulfill the request, it is put into the *wait container*. The *event processor* takes events from the event container, matches them with requests in the wait container and writes matching requests to the request container (*rescheduling*). The *timeout handler* regularly checks the *wait container* for requests that have timed out and moves them to the *request container*[1]. The *receiver* and the *sender* are responsible for handling remote requests and responses. The *embedded API* handles the access to the embedded and remote spaces.

Figure 3.3 is taken from [Sch08a] where XcoSpaces is described. The model used for MozartSpaces 1.0 in [Sch08b] is practically the same, just the event processor is termed differently with *wait* (component). The containers in the runtime model are described as XVSM containers and the according operations (write, take) are used. In the actual implementations different data structures are used for performance reasons, and other parts of the runtime are optimized as well.

In the MozartSpaces implementation an unbounded thread pool is used for the XP. Tasks that process requests are executed in its worker thread tasks. The queue of the XP thread pool is the request container, there is no response container. There is also no explicit event container and event processor. The event handling is very

[1]The timeout is also checked by the XP, where the timeout error response is created

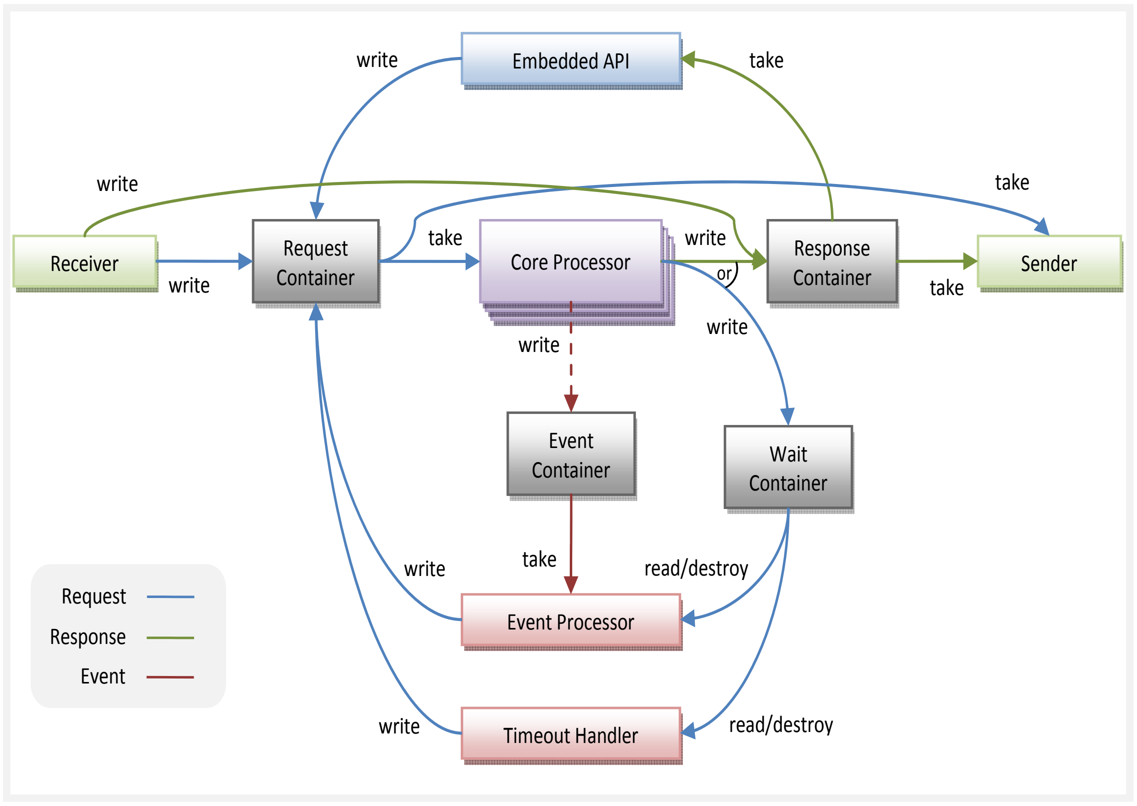


Figure 3.3: Runtime model of the previous XVSM version [Sch08a]

simple. A list of waiting request tasks is stored in each container and all tasks are rescheduled after an operation affected that container. To process timeouts, there is a separate timeout handler instance for each container, with entry requests for this container, and one additional instance for the transaction timeouts. All timeout handlers together use a thread pool for the periodic check for timed out requests and transactions. A third thread pool is used by the sender and receiver for remote communication. The embedded API is represented by the class *Capi* and allows only synchronous calls.

In the XcoSpaces implementation the thread pool provided by the .NET Framework is used for the XP. For the benchmarks in [KMM$^+$08] the asynchronous programming library Microsoft Concurrency and Coordination Runtime (CCR) with an internal thread pool was used. These thread pools also have internal queues that are used instead of an explicit request container. Like in MozartSpaces there is no response container. However, the event handling is more sophisticated and implemented like in the model. The event processor runs in its own thread, takes event for event from the event container and processes it. The matching of waiting requests and events compares the container, the operation/event type and in special cases also the transaction. The timeout handler is implemented with .NET timers.

The sender and receiver components are combined in a *communication service*. The embedded API allows only synchronous requests.

3.3 Formal Model

This section outlines the formal model as described in [Cra10], except where otherwise noted. It has already been implemented in the functional programming language Haskell and is also the basis for MozartSpaces 2.0. First we describe the fundamental data structures and operations on it, then the actual space operations with a layered architecture.

3.3.1 XVSM Algebra

The algebraic basis for an XVSM space is a hierarchic collection of data objects and other collections. The data objects and collections have a *label* that identifies them. A collection can be a *sequence* (list), that has an order, or a *multiset* (bag), when the order is not relevant. The basic data structure is an *xtree*, which is defined in [Cra10] as "either a sequence or a multiset of labeled xtrees, or an unstructured value like a string or an integer". A multiset of labeled xtrees is represented as $[l_1{:}x_1, l_2{:}x_2, \ldots]$ and a sequence as $\langle l_1{:}x_1, l_2{:}x_2, \ldots\rangle$. Multiset and sequence operations can be used on xtrees. A *path*, consisting of the labels from the root downwards separated by slashes, is used to access a sub-xtree. Items in a collection can have the same label. In that case one item is selected indeterministically. For sequences the item can be specified with an index number, but for multisets the ambiguity remains. A sequence xtree could be (taken from [Cra10] as the following xtree, entry and query examples)

$$X = \langle \texttt{abc}, \texttt{a}{:}24, \texttt{a}{:}42, \texttt{b}{:}[\texttt{pi}{:}3.14, \texttt{e}{:}\text{“text”}], [\ ]\rangle$$

and accessed with a path like in

$$X.(\texttt{b/e}) = [\texttt{pi}{:}3.14, \texttt{e}{:}\text{“text”}].(\texttt{e}) = \text{“text”}.$$

An XVSM *space* with its *containers* is defined as a multiset xtree. A container has a unique label and is itself a multiset xtree, containing the *entries* with a unique label. An entry consists of *properties*, label-value pairs where the value can be unstructured or an xtree. Entries are user-defined, a book could be represented, for example, as follows:

$$E_1 = [\texttt{title}\text{:“JavaSpaces”}, \texttt{author}\text{:“Freeman”}, \texttt{author}\text{:“Hupfer”}, \texttt{author}\text{:“Arnold”}, \\ \texttt{date}\text{:1999}, \texttt{publisher}\text{:“Addison-Wesley”}, \texttt{nPages}\text{:368}]$$

In addition to this user data there is also meta data in a space. It is used internally and can be stored at any level of the above described space xtree. Examples for meta data are not yet processed requests, transaction locks or coordinator properties. Besides the user containers there are also internal system containers which contain such meta data. An overview of the internal meta data gives the XVSM *meta model* in [Cra10]. The user and meta data together represent the complete space at runtime. Several spaces together form the XVSM *universe*, where each space is identified by a unique URI as label. Up to now, no more comprehensive construct, like, for example, a *multiverse*, has been defined.

3.3.2 Query Language

The XVSM Query Language (XVSMQL) has been introduced to allow more efficient access to sub-xtrees than by specifying a path. An XVSM Query (XQ) consists of one or more Simple XQs (SXQ) which are chained together, that is, the output of an SXQ is the input for the next SXQ. Each SXQ filters the input xtree; only properties of the xtree that fulfill it are part of the output xtree and no data is added.

SXQs are partitioned into *matchmakers*, which are predicates that can be evaluated on a single element and evaluate to true or false, and *selectors*[2], which operate on the whole input xtree. There are predefined matchmakers, e.g., for common relational operators ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$) and logical operators ($\wedge$, $\vee$, $\neg$) to combine matchmakers. Selectors are predefined for selecting a sub-collection of `n` items with `cnt(n)` (`cnt` is short for *count*), sorting collections with `sortup(p)` and `sortdown(p)` for the property values at path `p`, revert a sequence with `reverse()`, and ensuring uniqueness with `distinct(p)`. Additionally, wildcards (*) are allowed in a path for a query to select properties. A combined query of three SXQs for the entry example from above could be

$$(\langle E_1 \rangle \mid \text{*/author} = \text{“Hupfer”} \ \wedge \ \text{*/nPages} > 300 \mid \textit{sortup}(\text{date})) = \langle E_1 \rangle.$$

This query selects all entries with “Hupfer” as one of the authors and more than 300 pages from the the input sequence and sorts these entries in ascending order

[2]The selectors in XVSMQL are different from the coordinator-specific selectors to select entries from a container with a selecting operation.

by date. The input sequence contains (only) the example entry from above, which fulfills this query and is thus present in the output sequence.

3.3.3 Layered Architecture

Based on the algebra and query language described above, XVSM is defined with a layered architecture. Figure 3.4 shows the different layers that build on each other. CAPI-1 provides basic atomic operations to access xtrees. CAPI-2 introduces transactions where several operations are grouped to one atomic action. CAPI-3 adds support for containers with user definable coordinators. These three layers support already most of the data storage functionality of the space, but their operations are synchronous and non-blocking. If an operation cannot be fulfilled, for example, reading entries because the container is empty, the operation returns immediately and indicates that in the result status (`DELAYABLE` in this example). In general, the following status codes are defined:

- `OK`: The operation finished successfully.
- `NOTOK`: An error occurred, e.g., because of invalid argument values.
- `DELAYABLE`: The operation cannot be fulfilled at the moment.
- `LOCKED`: The operation was not fulfilled because a locked data structure was encountered. This status is only returned by CAPI-2 and CAPI-3.

The runtime in CAPI-4 manages the timeout of operations and schedules the request processing. It also initializes the meta model where system containers are used. No coordinators or transactions are necessary to access these system containers, but the CAPI-1 operations alone are not sufficient, because they are non-blocking. Therefore CAPI-B with blocking variants of the CAPI-1 operations was added. The runtime also allows the extension of space operations with so-called aspects. On top of it is the language independent XVSMP. The XVSM semantics is defined at this layer. Language bindings that send and receive XVSMP messages can be created above it. The runtime is called asynchronously by the XVSMP layer or an embedded API.

The strict CAPI layering from the formal model is implemented in the Haskell prototype. In MozartSpaces the layers CAPI-1, CAPI-2 and CAPI-3 are combined for performance reasons. However, the same functionality is provided by the interface of CAPI-3.

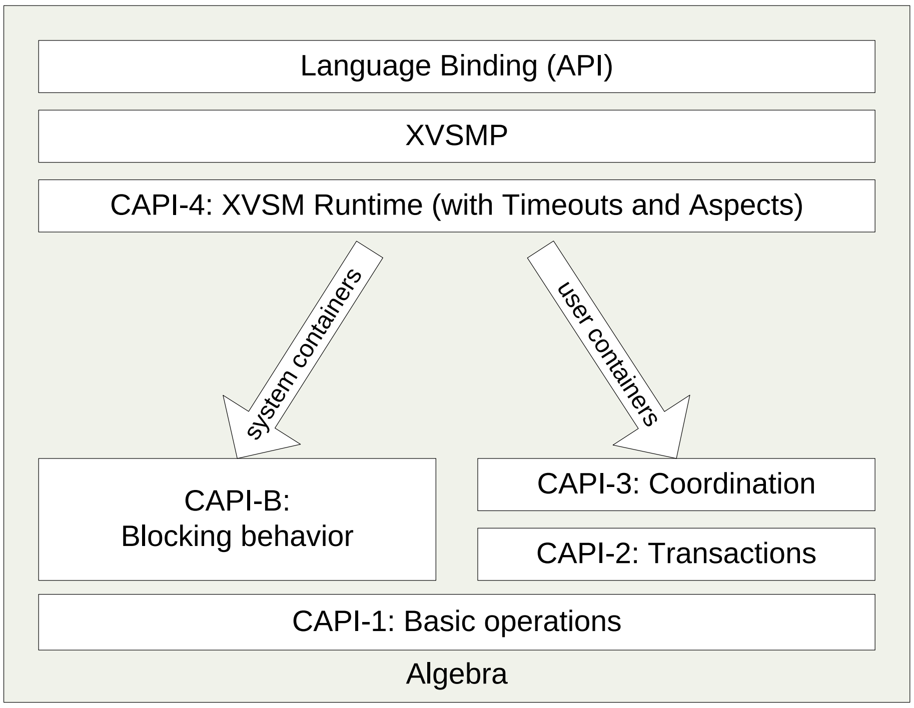


Figure 3.4: Layered architecture of XVSM [Cra10]

3.3.4 Basic Operations, Transactions and Coordination

The basic atomic operations of CAPI-1 can be used to write an xtree at a specified path and read or take an xtree with a path and query.

The transaction model of CAPI-2 has transactions and sub-transactions. Transactions can be controlled by the user, sub-transactions are used in the runtime to encapsulate the processing of a single (user) request. A sub-transaction is started by the runtime at the beginning of the processing of a transactional request and can span several CAPI-3 operations, though there is usually only one. The sub-transaction is committed when the request processing was successful (status `OK`), otherwise it is rolled back. XVSM uses pessimistic concurrency control, that is, xtrees are locked when they are accessed. There are different types of locks depending on the operation that is used for access. Written xtrees have an insert lock and are invisible for other transactions. Taken xtrees have a delete lock, the visibility depends on the isolation level. For the isolation level `REPEATABLE_READ` it is exclusive, so other transactions cannot access the xtree. For the isolation level `READ_COMMITTED`, implemented in CAPI-3 of MozartSpaces 2.0 [Bar10], other transaction may read but not take delete-locked xtrees. When xtrees are read, shared (non-exclusive) read locks are acquired for the isolation level `REPEATABLE_-`

`READ`. For the isolation level `READ_COMMITTED` no read locks are used. Also important is the visibility between transactions and their sub-transactions. Concurrently active sub-transactions are isolated like different transactions. However, sub-transactions can access data locked by the parent transactions. When a sub-transaction is committed, all its locks are handed over to the parent transaction. To actually insert or remove xtrees and remove the acquired locks on commit or rollback, a transaction log with items for each transactional operation is kept. CAPI-2 offers operations to create, commit and rollback transactions and sub-transactions, which are used by CAPI-4. The CAPI-2 interface contains also transactional operations to write, read and take xtrees. Additional operations exist to set exclusive locks, an additional lock type, and retrieve readable and takeable properties at a path without locking them.

In CAPI-3 we have containers to structure the space xtree. They contain entries which are also xtrees. Optionally the containers can have a name that is unique in the space. The CAPI-3 interface provides operations to create and destroy containers and lookup named containers. These methods are transactional and an additional method to exclusively lock a container for a transaction is provided. Coordinators are specified when a container is created. There are two lists for them, one for the obligatory and one for the optional coordinators. Obligatory coordinators manage every entry in the container. In contrast, optional coordinators manage only the entries for which they are specified with coordination data when they are written to the container. CAPI-3 also offers transactional operations to write entries to a container, and read, take or delete entries from a container. When entries are written to a container, coordination data[3] corresponding to the container's coordinators have to be specified. They can pass entry-specific information to a coordinator. When entries are read, taken or deleted, one or more read selectors have to be specified. If more than one read selector is specified, they are executed consecutively, like SXQs in XVSMQL, and only entries matching all selectors are returned.

There are several predefined coordinators which can be distinguished by their selector functionality: the System coordinator, which just returns the specified number of entries, without any specific order. The FIFO and LIFO coordinator provide queue ordering and stack functionality, respectively. The Linda coordinator allows for template matching similar to the Linda model. The Query coordinator supports XQs on entries. A unique key (value) can be used to access entries with the Key coordinator, for the similar Label coordinator a non-unique label is used. With the

[3]The term *write selectors* is used in [Cra10]. Later we decided to replace it with the term *coordination data*, because entries are not selected when they are written to a container and we want to avoid confusion with the "read" selectors.

Vector coordinator entries are accessed with their numeric index.

Compared to the original formal model, some changes regarding coordinators were made to CAPI-3 of MozartSpaces during the development of the runtime. In [Bar10] the same classes are used for read and write selectors. We changed that and now distinguish coordination data for write and (read) selectors for selecting operations. In addition to obligatory and optional, coordinators can now also be *implicit* or *explicit*. For implicit obligatory coordinators no coordination data needs to be specified when entries are written, the entries are automatically bookkeeped by them. Implicit optional coordinators need to be addressed with according coordination data. Explicit coordinators always need entry-specific coordination data when an entry is written. The predefined coordinators Any, FIFO, LIFO, Linda, Query and Random are implicit coordinators, Key, Label and Vector are explicit coordinators. The Any coordinator is a new coordinator that has the coordination functionality of the System coordinator and returns entries without a specific order. The Random coordinator, already specified in [Bar10], is very similar, but shuffles the entries before returning them.

3.3.5 Runtime Model

The runtime model is depicted in Figure 3.5. It is similar to the runtime model of the previous XVSM version (cf. Figure 3.3), mainly the event processing looks different. There is also a request container where user requests are written by the embedded API and the receiver (green arrow), or taken by the sender for processing on a remote core (blue arrow). The XP concurrently takes requests from the request container and processes them. The result is written to the response container where it is retrieved by the embedded API or a sender for a request from a remote core (green arrow). For a request processed on another core the result can also be written to the response container by the receiver (blue arrow).

The XP processes requests by calling the appropriate CAPI-3 operation and aspects (explained below). This leads to a result with a status code as explained in Section 3.3.3. When the status is `OK` or `NOTOK`, the request is completely processed and a response with the result or error, respectively, is written to the response container. When the status is `LOCKED` or `DELAYABLE`, the request is handed to the event processing (see below), where it may be put into the wait container, if its timeout value allows it. The timeout value can be specified by the user and influences whether and for how long a request is blocked. It may be one of the following values:

- Integer > 0: This specifies the time in milliseconds, starting with the time when it is taken from the request container by the XP, after which a timeout

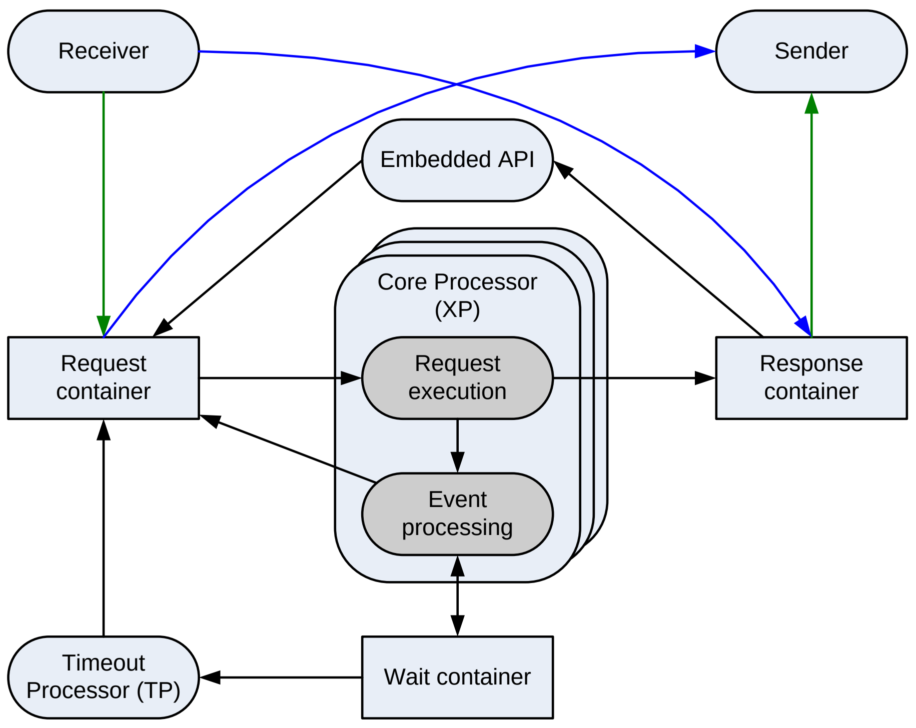


Figure 3.5: XVSM Runtime in the formal model [Cra10]

error response is created by the XP when the request is not yet completely processed and rescheduled by the timeout processor (see below).

- ZERO: The request is processed once, that is, the according CAPI-3 operations is called only once, and it is never rescheduled. An error is returned for the status `LOCKED` and `DELAYABLE`.
- INFINITE: The request does not expire, it may indefinitely be blocked and rescheduled as long as no result is available and no error occurs.
- TRY-ONCE: The request is processed once on the actual data with all necessary locks acquired. The request is blocked when CAPI-3 returns `LOCKED`, an error is returned for the status `DELAYABLE`.

The event processing described below may put a request into the wait container. The timeout processor periodically checks for timed out requests in the wait container and reschedules them. As the timeout is checked by the XP at the beginning of the request processing, for such requests an error response is created subsequently by the XP.

Event Processing

XVSM has a blocking behavior for certain requests, i.e., they can be blocked when they are not successfully processed by CAPI-3 or aspects, and return with status `LOCKED` or `DELAYABLE`. Blocked requests are put into the wait container. They are rescheduled, that is, put again into the request container, when matching events occur. For example, a read request on an empty container may return with `DELAYABLE` and is rescheduled later when an entry is written to that container and the request can possibly be fulfilled. In general, blocked requests wait for an event of a specific category (type) that depends on the CAPI-3 result and the request type. When requests are processed, events of specific types that also depend on the CAPI-3 result and the request type are generated.

The event types `insert` (entry was written) and `remove` (entry was taken or deleted) are relevant for the CAPI-3 status `DELAYABLE`. The blocked read request from the example above would wait for `insert` and the write request would generate an `insert` event that matches the waiting read request and causes it to be rescheduled. As the wait categories and events for a specific request are fixed, an according coordinator behavior is expected in the event processing. For the status `LOCKED` it is relevant whether the request was locked because of a lock of a transaction or an active sub-transaction as explained in the following. A transaction can be further used after a sub-transaction of it is rolled back and a sub-transaction can set volatile locks (*short-term locks*) that are removed when it is rolled back. In contrast, *long-term locks* from an already committed sub-transaction are held until the transaction ends. We want to avoid unnecessary rescheduling of requests that are blocked because of long-term locks by events generated due to the release of short-term locks. Accordingly, the categories `lt_unlock` when a request is locked because of a long-term lock, and `st_unlock` when a request is locked because of a short-term lock, are distinguished.

For the event processing in general we consider more information than the wait category and the event type. Figure 3.6 shows the data structure that is used in the wait container. We distinguish the wait categories `insert`, `remove`, `lt_unlock` and `st_unlock` for each container. Inside of a wait category are the waiting requests, depicted with the information relevant for the event processing, and additional event timestamps that are independent of waiting requests. The *last committed time* is the last time that a transaction with operations on that container committed (or a sub-transaction with a status that is not `OK` was rolled back). The *uncommitted times* are transaction-specific timestamps that indicate when events for that transaction were processed the last time. The timestamps are necessary to prevent race conditions that can otherwise occur because of the concurrent request and event processing by the XP. The transaction-specific timestamps are used to consider the transaction visibility for the event processing and avoid to reschedule requests unnecessarily. For a detailed description see [Cra10]. While the event processing compares the container, category, transaction and timestamps before a task is rescheduled, it can still happen, that a request is rescheduled but can still not be fulfilled. An example would be a blocked read request that selects two entries and is rescheduled after a write request has written one entry to an empty container. By making the event processing more fine-grained that could be prevented, but there is a trade-off between the computational effort and complexity of the request processing and the cost of unsuccessfully processing a request.

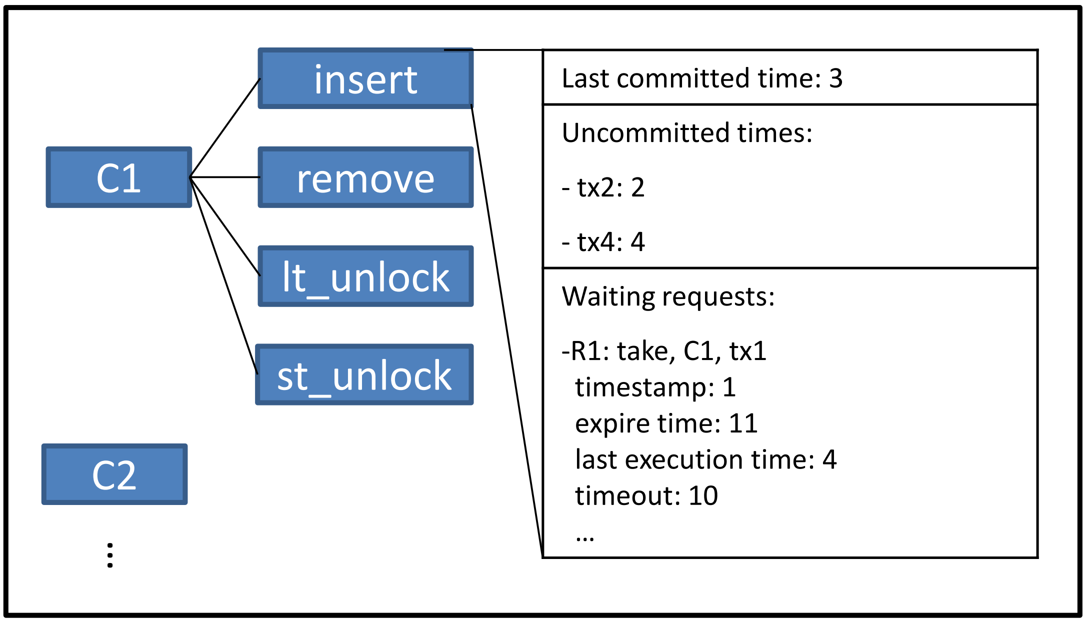


Figure 3.6: Data structure for waiting requests and event timestamps

Aspects

Aspects in XVSM are code segments that can be added dynamically during runtime and are executed when a request is processed by the XP. The runtime provides operations to add and remove aspects for so-called *interception points* (ipoints). There are two ipoints for each request type, before (pre) and after (post) a request is processed by the XP. Aspects are executed in the defined order. They can access and modify the request (pre-aspects) or the result (post-aspects). They are an implementation of the interceptor pattern that is also used for distributed object middleware [SSRB00]. Aspects can be *local*, that is, only defined for a specific container, or *global* and executed for every space operation. They can use the same sub-transaction that is used for the request processing and access the space or external system inside of it. However, changes outside of the space are not automatically committed or rolled back. Aspects can return the same return status codes as CAPI operations (see Section 3.3.3). If an aspect returns with a status code that is different from `OK` the execution of the aspects is aborted and this status is returned instead of the actual CAPI-3 operation status. Pre-aspects can also return `SKIP` to skip the execution of the actual CAPI operation. Every aspect can access the so-called *request context* of the request, which is specified in the user API. Thereby, arbitrary data can be passed to aspects, if necessary.

Overall, there is no exact definition of the aspects in the current formal model. A simple implementation of them is described in Section 5.2. This implementation is similar to the aspects in the previous XVSM implementations. Java classes that implement a defined interface can be used as aspects. An extension could be to support aspects written in an interoperable scripting language or allow for a more flexible definition of interception points [KS05].

CAPI-4 Operations

The formal model defines CAPI-4 operations that are indirectly called by writing a corresponding request into the request container. All operations have a space URI, that specifies where the request shall be processed, and a (request) context as parameter. There are operations to add and remove an aspect. Operations to create, commit and rollback a transaction can be used for transaction control. A timeout can be specified when a transaction is created. CAPI-4 also provides operations to create, destroy, lookup and lock a container. Further operations allow to write entries to a container, and read, take or delete entries from a container. A timeout can be specified for the entry operations. Entry and container operations are transactional and an explicitly created transaction can be passed to them. If no transaction is specified, the runtime creates an transaction internally. This *implicit transaction* is used like an explicit transaction but is automatically committed when the request returns with a result, or rolled back when the request returns with an error.

3.3.6 XVSMP and Language Bindings

XVSMP is the interoperable protocol that defines the XVSM semantics and can be used for remote access to a space. It is not specified in detail for the new formal model but essentially comprises the CAPI-4 operations from above with the defined semantics. The protocol describes the representation of the CAPI-4 request xtrees and the corresponding responses as messages in an interoperable format, for example XML. These messages can be asynchronously sent and received by the sender and receiver of the runtime. Usually the response to a request is delivered to the space where the corresponding request was issued. This is the standard request-response message exchange pattern. A so-called *answer container* extends the request message with an alternative destination for the response. This answer container can be a user container on any space. In the request-response case the answer container is *virtual*, it is the API that was used to send the request. This API can be the embedded API (cf. Figure 3.5) or a language binding. Such a language binding can provide an XVSM API for an arbitrary programming language. It translates API calls to XVSMP request messages, sends them and receives response messages either synchronously or asynchronously with callbacks. In [KRL07, KRML08, Lec08] a JavaScript API for XVSM is described. Although this API is for an older version of XVSM, a similar implementation could be used for the new formal model and its XVSMP.

3.4 Profiles

Profiles are extensions of the XVSM *core*. They can have their own API above the embedded API or XVSMP, and extend the core functionality with *modules*. Modules can be aspects, custom coordinators, additional transport protocols or other extensions that are plugged into the core. An example are notifications that implement the publish-subscribe model [EFGK03]. Notification support can be added to XVSM with aspects [KMK^{+}09, KMS08a]. One form of notifications has been implemented for MozartSpaces 2.0 and is described in Section 5.2.5. This profile uses aspects and provides a simple notification API that complements the core API. Internally, existing requests are used and thus XVSMP is not extended. Further profiles include distributed lookup, replication, security extensions for authentication and authorization, life cycle management, integration adapters and administrative extensions for configuration and monitoring. Some profiles have already been developed for MozartSpaces 1.0. For example, [Goi09] describes distributed lookup and several protocol plug-ins.

4 MozartSpaces Runtime Architecture

The last chapter described XVSM and the new formal model that is the basis for the XVSM implementation MozartSpaces. This chapter describes the architecture of the MozartSpaces 2.0 runtime that was implemented in the course of this thesis. Section 4.1 starts with an overview before Section 4.2 describes the XP and Section 4.3 the blocking behavior of the runtime. Section 4.4 explains the internal configuration of the runtime.

4.1 Overview and Components

As described in Section 3.3.5, the runtime provides an environment for the concurrent execution of coordination operations. It adds support for blocking operations with timeouts and manages and invokes aspects. It corresponds to CAPI-4 in the layered architecture of the formal model and uses the CAPI-3 implementation described in [Bar10]. The *core API* (see Chapter 6) provides embedded access to spaces and the *sender* and *receiver* components (see Section 5.3) are responsible for the remote access.

Compared with the formal model, several changes were made to the internal structure for the implementation of the runtime. The reason is, that some parts of the model are difficult to implement with high performance. This was already experienced with the previous implementations and is explained in [Sch08a]. It also affected the CAPI-3 architecture. CAPI-3 is internally not separated into the different layers and there is no CAPI-1 interface. Accordingly, there is no CAPI-B in MozartSpaces (cf. Figure 3.4) and the runtime cannot bootstrap itself with own mechanisms. So the runtime uses no XVSM containers. None of the runtime containers in the XVSM meta model actually exists in MozartSpaces. The *request*, *response* and *wait containers* are implemented mainly with standard Java collections as data structures that are not accessed with CAPI operations.

4.1.1 Runtime Structure

Figure 4.1 shows the structure of the runtime, especially the data flow from outside the runtime to the XP and back. The rectangles are components and the arrows between them are marked with the types of the objects that are passed from one component to the other. Compared to the runtime structure of the formal model, we see that the *request container* is now named *request handler* and there is the *response distributor* instead of the *response container*. This names should resemble their functionality more accurately because they are not containers in the implementation. Furthermore, the *message distributor* and the *response handler* process receive messages and the *response distributor* passes responses to the *core API* or the *sender*. The *wait container* is now named *wait & event manager* and includes the event processing. The requests and responses in Figure 4.1 are all encapsulated into respective messages that contain some additional information.

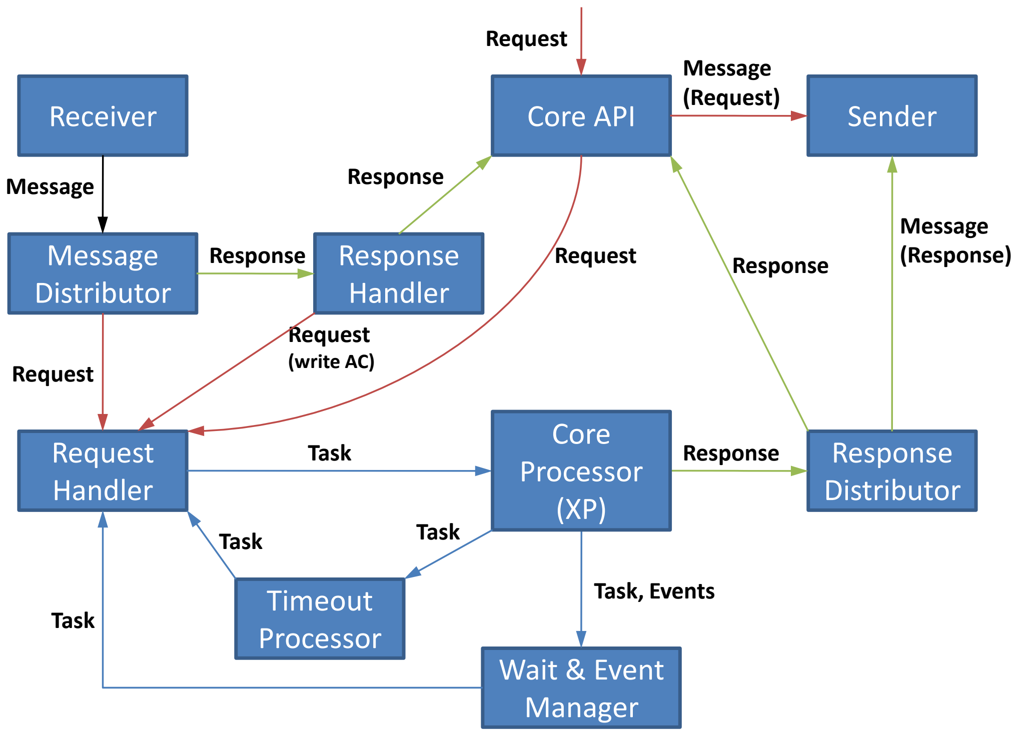


Figure 4.1: MozartSpaces runtime structure

4.1.2 Messages, Requests, Responses and Tasks

There are objects of different types passed between the runtime components. In general, the runtime processes *request objects* and returns a *response object* for each request object. These objects are encapsulated into messages that contain a unique request identifier and routing information.

Figure 4.2 shows the message interface and classes. The *request reference* is the unique identifier of a request, the *content* the actual request or response and the *answer container information* defines the destination of the response. If it is not set, the response is sent back to the virtual answer container of the core where the request has been created. If it is set, it defines the container where the response should be written to. The response is then written with a request that is created by the response handler (see Figure 4.1). The answer container information is also the reason for the difference between the request and response message classes regarding the URI of the destination space. For a `RequestMessage` the destination space URI is explicitly set. For a `ResponseMessage` it can be derived from either the request reference, when the answer container information is not set, or the reference of the answer container.

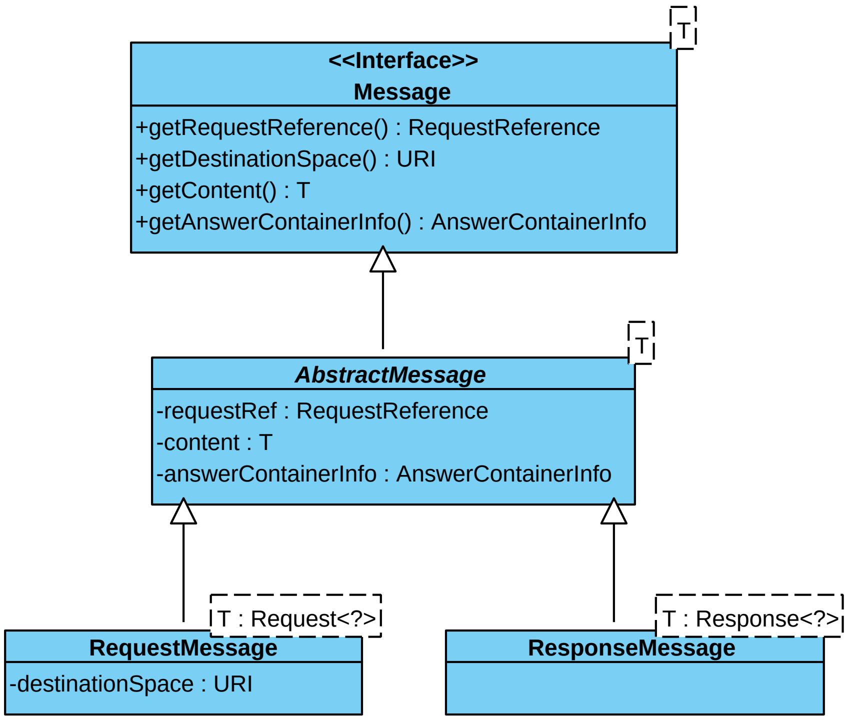

T
<<Interface>>
Message
+getRequestReference() : RequestReference
+getDestinationSpace() : URI
+getContent() : T
+getAnswerContainerInfo() : AnswerContainerInfo
T
AbstractMessage
-requestRef : RequestReference
-content : T
-answerContainerInfo : AnswerContainerInfo
T : Request<?>
RequestMessage
-destinationSpace : URI
T : Response<?>
ResponseMessage

Figure 4.2: Message interface and classes

Figure 4.3 shows the hierarchy for the request classes. There is a concrete class for each different type of operation to be performed on the space, because they have different properties as arguments. How these properties are used in the runtime is explained in Section 4.2.3. Many properties have a default value, others need to be specified when a request is created with an API call as explained in Section 6.1.

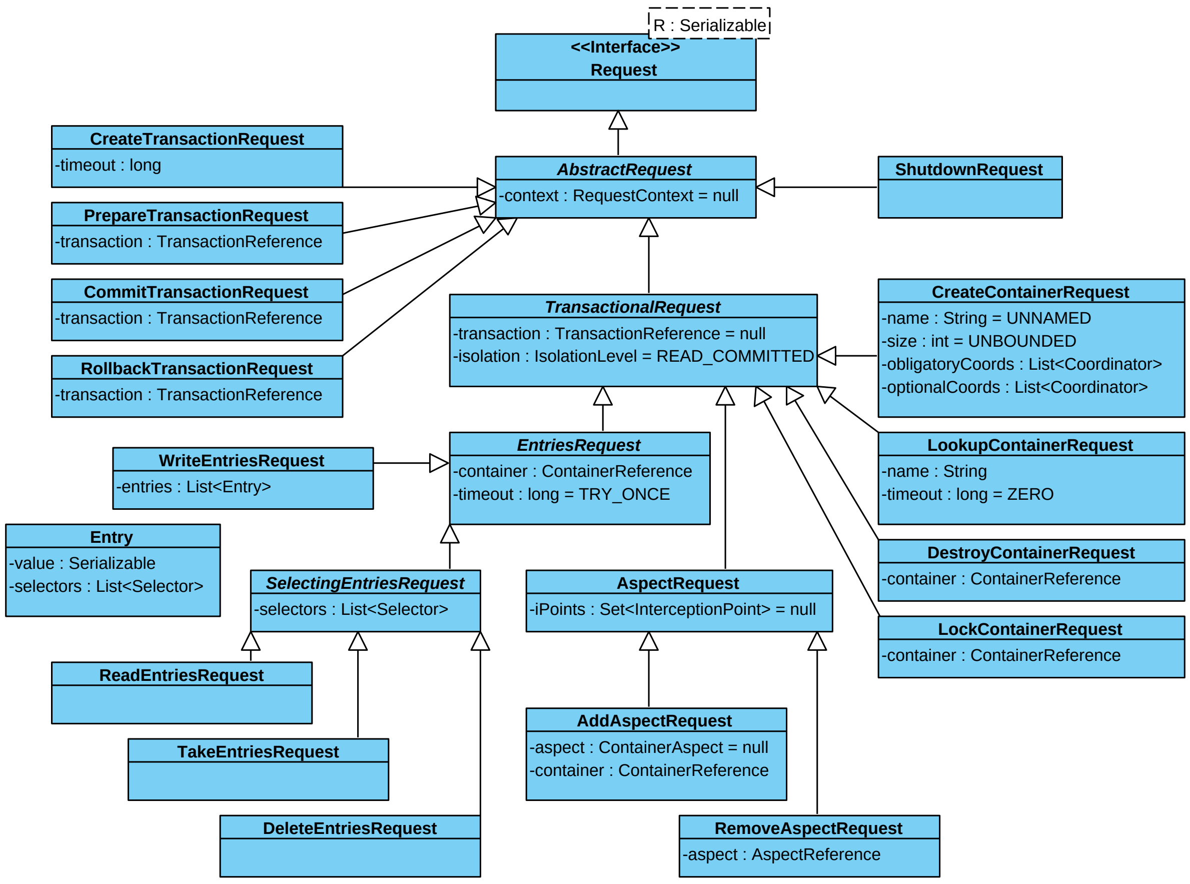

R : Serializable
<<Interface>> Request
AbstractRequest
-context : RequestContext = null
ShutdownRequest
CreateTransactionRequest
-timeout : long
PrepareTransactionRequest
-transaction : TransactionReference
CommitTransactionRequest
-transaction : TransactionReference
RollbackTransactionRequest
-transaction : TransactionReference
TransactionalRequest
-transaction : TransactionReference = null
-isolation : IsolationLevel = READ_COMMITTED
CreateContainerRequest
-name : String = UNNAMED
-size : int = UNBOUNDED
-obligatoryCoords : List<Coordinator>
-optionalCoords : List<Coordinator>
LookupContainerRequest
-name : String
-timeout : long = ZERO
DestroyContainerRequest
-container : ContainerReference
LockContainerRequest
-container : ContainerReference
WriteEntriesRequest
-entries : List<Entry>
EntriesRequest
-container : ContainerReference
-timeout : long = TRY_ONCE
Entry
-value : Serializable
-selectors : List<Selector>
SelectingEntriesRequest
-selectors : List<Selector>
AspectRequest
-iPoints : Set<InterceptionPoint> = null
ReadEntriesRequest
TakeEntriesRequest
DeleteEntriesRequest
AddAspectRequest
-aspect : ContainerAspect = null
-container : ContainerReference
RemoveAspectRequest
-aspect : AspectReference

Figure 4.3: Class hierarchy for requests

Figure 4.4 shows the response interface and the generic implementation that is used for the responses of all request types. The result type depends on the request type and is also explained in Section 4.2.3.

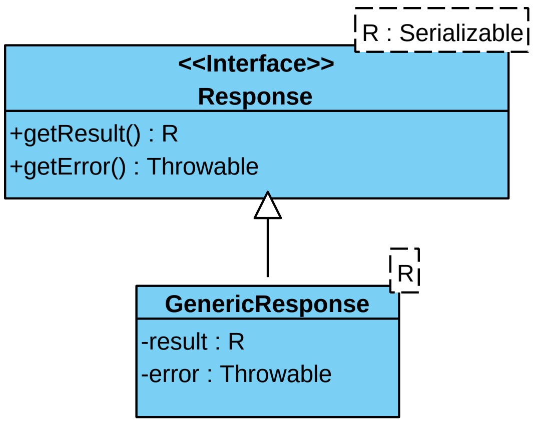


Figure 4.4: Class hierarchy for responses

A *task* is created for each request message that is handed over to the request handler. There are different task types, one for each request type, as it is explained in Section 4.2.2. A task execution eventually leads to a response, often also to *events*.

4.1.3 Data Flow between Components

In Figure 4.1 we can see that there are two ways how a request can enter the core, through the core API or encapsulated in a *message* from a *receiver*. There is also the special case that a write request is created for a response by the *response handler*, when the response should be written to the space because an answer container has been specified for the original request. In all cases the request is routed to the *request handler*. The request handler creates a *task* for the request, which is executed in the *XP*. If the task execution can completely process the request, a *response* is created. Otherwise, the task is stored in the *wait & event manager*. In this case the task is additionally put into the *timeout processor*, if a positive timeout value is set. The execution of a task that accesses CAPI-3 generates events that are processed in the wait & event manager. A task in the wait & event manager is rescheduled and executed again when matching events are generated. For each request exactly one response is created. The response is passed to the *response distributor* and, depending on its origin and whether an *answer container* is set, forwarded to the embedded core API or the *sender*. A response message sent from another core is received by a MozartSpaces receiver and passed to the *response handler*. The response it contains is either destined for a specific request in the core API or an answer container on that space. In the latter case, the response handler creates a request from the response content, a standard *write request*, and passes it to the request handler.

4.2 Core Processor (XP)

The XP is the part of the MozartSpaces runtime where the requests are actually processed. This section describes the tasks and components used for it.

4.2.1 XP Structure

The XP is basically a thread pool inside the request handler where tasks are executed. A task is a `Runnable` object that accesses CAPI-3 and runtime components (transaction manager, aspect invoker, aspect manager) to process a request. It is executed in a worker thread of the thread pool. Figure 4.5 shows this overall structure which is a more detailed view of the XP in Figure 4.1. There are different types of tasks for requests, which is explained in the next section.

A thread pool consists of a number of threads that execute tasks, so called *worker threads*, and code to manage these threads and the task execution. Thread pools provide better performance and manageability than a bunch of single threads. The creation and destruction of threads is computationally expensive. The worker threads in the pool are reused, that is, they execute task after task and thereby mainly avoid the creation/destruction overhead. The tasks are taken from a queue that is shared by all worker threads. Which thread eventually executes it is usually decided by the thread pool internally. The Java API provides different thread pools that can be instantiated and are ready to use with one statement. Section 4.4.2 describes which thread pools can be configured and used in the runtime and discusses their advantages and disadvantages.

4.2.2 Request Task Creation

The requests that are processed in the MozartSpaces runtime can be of different types and there is a concrete task class for each request class. The type hierarchy for the tasks is shown in Figure 4.6 and is similar to the type hierarchy for the requests in Figure 4.3. When a request is passed to the request handler, a task is created for it—for example, a `CreateTransactionTask` for a `CreateTransactionRequest` or a `ReadEntriesTask` for a `ReadEntriesRequest`. The request and its properties are accessible from within the task, this information is used when the task is executed.

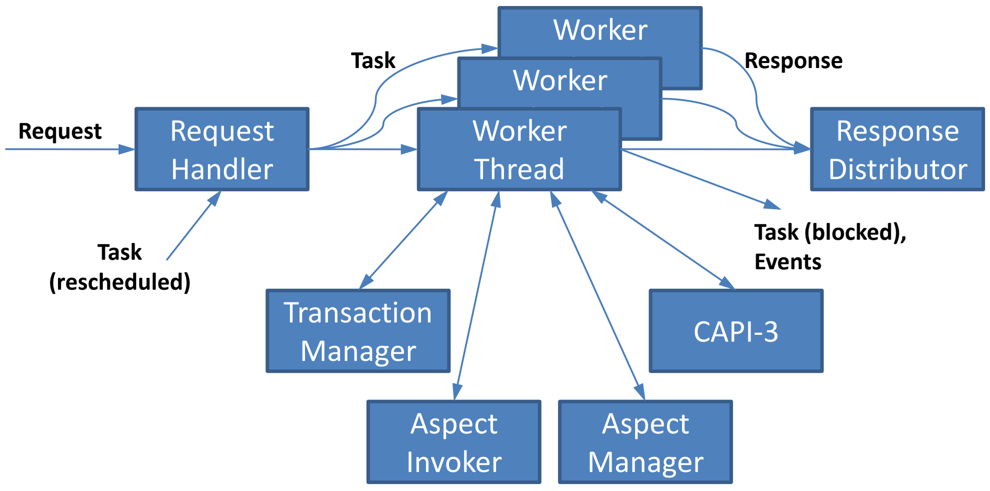


Figure 4.5: Core processor structure

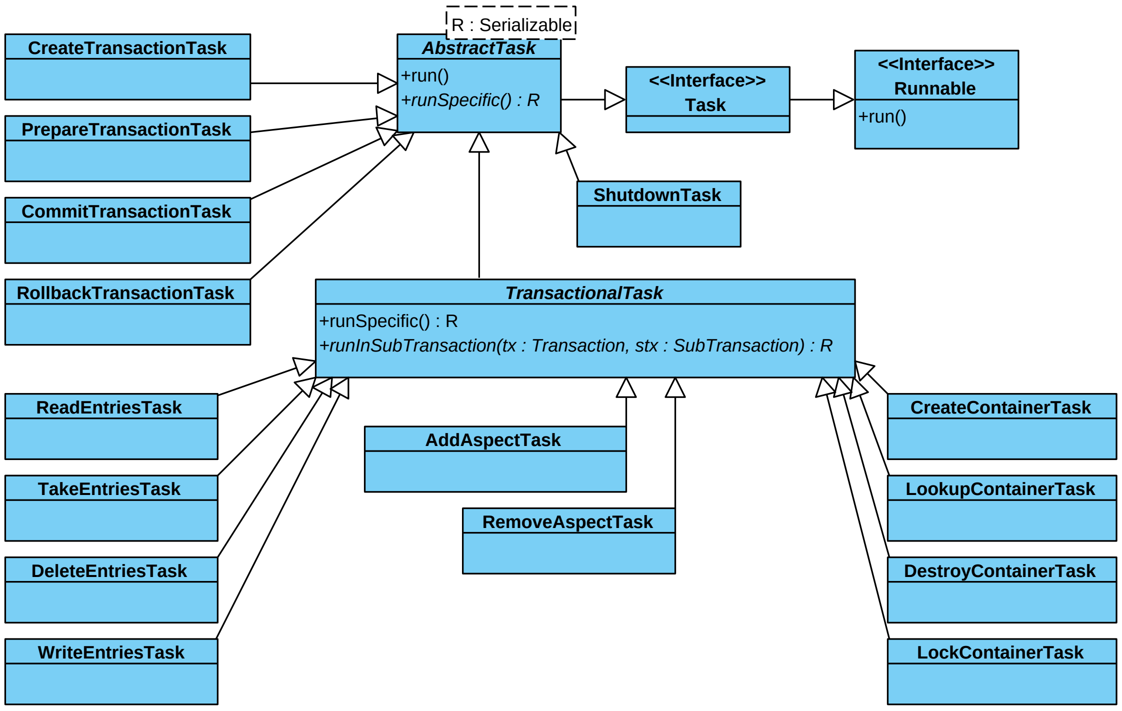


Figure 4.6: Request task hierarchy

4.2.3 Request Task Execution

The interface `RequestHandler` and its implementation `ThreadPoolRequestHandler` are shown in Figure 4.7. When the `ThreadPoolRequestHandler` gets a request, it internally creates a task for it and executes it with the thread pool, an instance of `ExecutorService`. By default this is done asynchronously, that is, the call to the `execute` method of the `ExecutorService` adds the task to the queue of the thread pool and returns immediately, before the task is executed. However, the details are at the discretion of the concrete thread pool and can be configured (see Section 4.4.2).

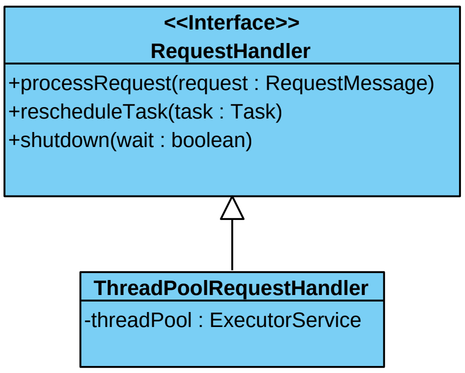

<<Interface>>
RequestHandler
+processRequest(request : RequestMessage)
+rescheduleTask(task : Task)
+shutdown(wait : boolean)
ThreadPoolRequestHandler
-threadPool : ExecutorService

Figure 4.7: Request handler

When a task is executed, the `run` method implemented in `AbstractTask` is called (cf. Figure 4.6). There the request timeout is checked. If it has been reached, a response with an error is sent, otherwise the method `runSpecific` is called. This method is either already specific for the concrete task or prepares the transaction and sub-transaction used in the concrete task, if the task is transactional. For such a `TransactionalTask` the method `runInSubTransaction` implements the main request processing. The request-specific part of the task execution is explained below. After `runInSubTransaction` the sub-transaction is committed, and also the transaction if it was created implicitly. For all tasks, when `runSpecific` returns, the request processing is either finished and a response with a result or error is sent, or the task is blocked, that is, it is passed to the wait & event manager. Subsequently, the wait & event manager is called to process the events generated by the request processing.

Example Task Execution

An example for a task execution is shown in Figure 4.8. A `ReadEntriesTask` processes a `ReadEntriesRequest` with an implicit transaction. In the `runSpecific` method the CAPI-3 implementation is called to create a new transaction, which is added to the *transaction manager*. The task type specific method `runInSubTransaction` is invoked with a new sub-transaction. In it the *aspect invoker* is called to execute the pre-aspects for read, then the actual read method of CAPI-3 and the post-aspects. Back in `runSpecific` the sub-transaction is committed and, because CAPI-3 returned the status `OK` and a result, the transaction is committed as well. At the end of the `AbstractTask`'s `run` method the response with the list of the read entries as result is handed to the response distributor and the wait & event manager is called to process the events.

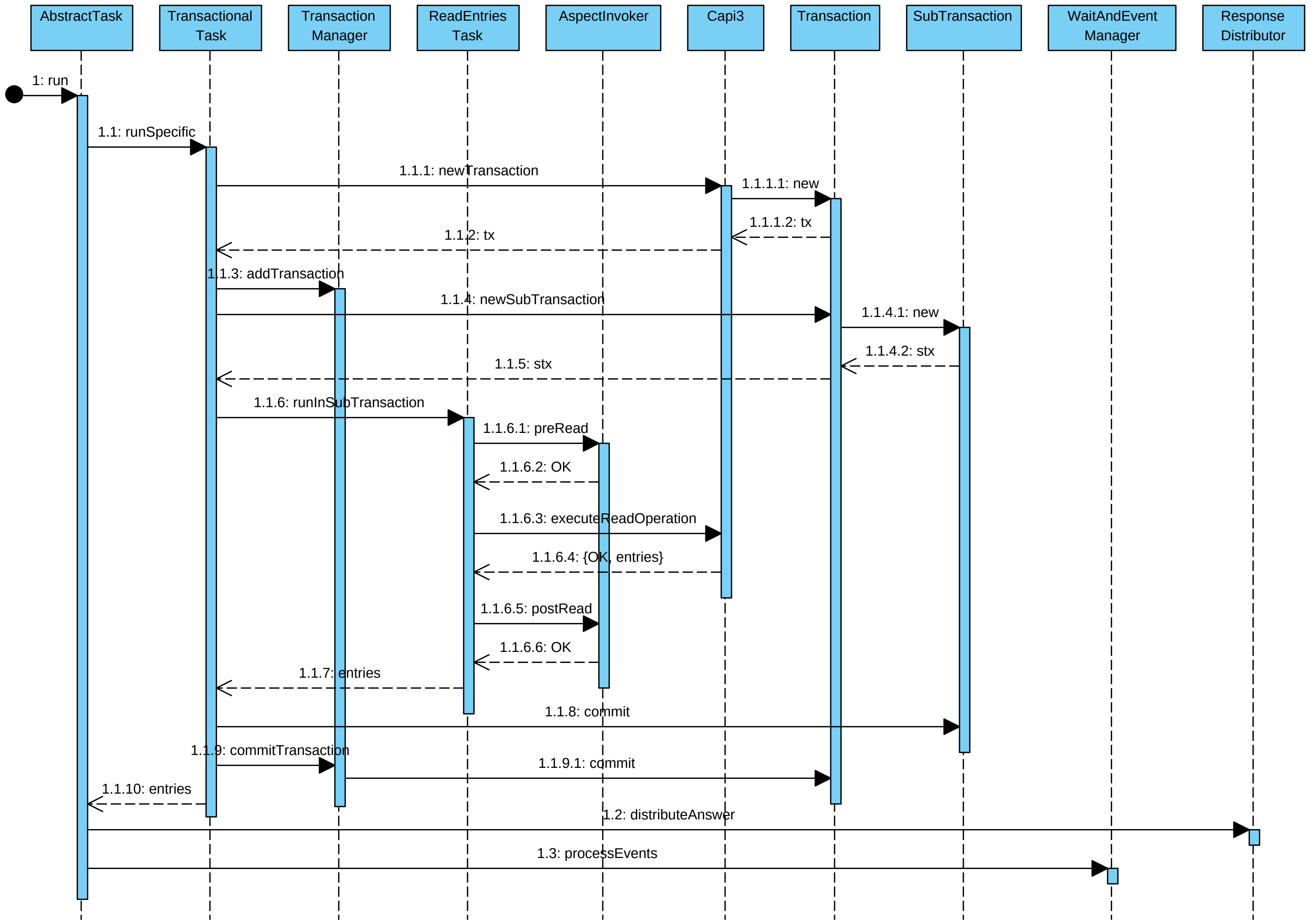


Figure 4.8: Example sequence for a `ReadEntriesTask` execution

Apart from error cases, the sequence for `ReadEntriesTask` would also be different if an explicit transaction was used or the CAPI-3 call did not return a result and thus the task was blocked. The differences for the other task types are explained below. For the types of the results returned by the different task types see also the description of the core API in Section 6.1. The aspect execution is described in Section 4.2.5.

Request Property Usage

The properties of the request are used when the task is executed. Some of the properties are specific for a request type, others are common for several request types and inherited from superclasses (cf. Figure 4.3). The request objects themselves are passed to the aspect invoker when the aspects are executed. The *context* property from `AbstractRequest` is passed to the CAPI-3 method called by the container and entry tasks. For `TransactionRequest`s, the transaction reference is used if it is set, otherwise a new transaction is created. The isolation level is currently only used in the container and entry tasks and passed to CAPI-3 like the context. The timeout properties of the entry tasks, the `CreateTransactionTask` and the `LookupContainerTask` are used by the runtime in the timeout processors. The usage of the other request properties are specific for a request type and explained below.

Container and Entry Tasks

The container and entry tasks are all very similar to the `ReadEntriesTask` in the example above. Different are the methods in the aspect invoker to execute the aspects, the CAPI-3 method that is called and the result type. The `TakeEntriesTask` returns a list of entries like the `ReadEntriesTask`. The two other entry tasks, `WriteEntriesTask` and `DeleteEntriesTask` return nothing. Regarding the container tasks, the `CreateContainerTask` and the `LookupContainerTask` return a container reference, `DestroyContainerTask` and `LockContainerTask` return nothing. Except for the timeout property, the container and entry task's specific request properties are just passed to CAPI-3 (see Section 4.2.6).

Transaction Management Tasks

The tasks to create, commit or rollback a transaction invoke the aspects and call the corresponding method of the transaction manager (see Section 4.2.4). The `Prepare-TransactionTask` invokes the aspects but does not use the transaction manager in `runInSubTransaction`, the preparation of a transaction is performed by calling a method of the internal transaction object. Regarding the return type, the `CreateTransactionTask` returns a transaction reference, the other tasks return nothing.

Aspect Management Tasks

The aspect tasks use the *aspect manager* to add or remove an aspect (see Section 4.2.5). The `AddAspectTask` returns an aspect reference, the `RemoveAspectTask` nothing.

Other Tasks

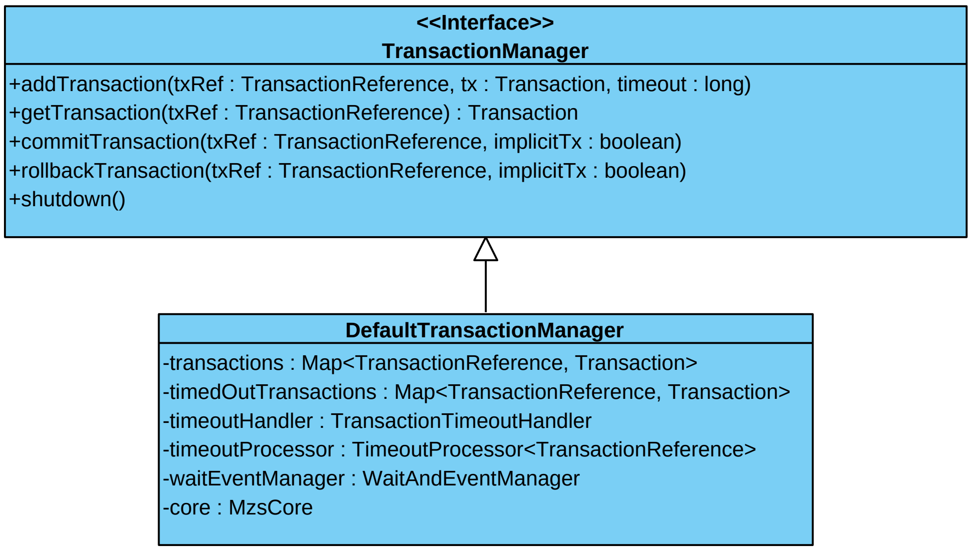
There are additional tasks to clear the space and shut down the core. The `ShutdownTask` initiates the shutdown of the core. The `ClearSpaceTask` (not depicted in Figure 4.6) does currently nothing because the meta-model that is needed to clear the space is not implemented yet. Both tasks return nothing.

4.2.4 Transaction Management

In XVSM, CAPI-3 is used to create and commit or rollback transactions, but the list of active transactions is kept in the runtime by the *transaction manager*. Its interface is shown in Figure 4.9. Whenever a transaction is created by a task, it is added with the method `addTransaction`. As we see in the interface, there is a parameter of type `TransactionReference` and one of type `Transaction`. The former is the external transaction representation, also used in the core API and in requests, while the latter is the internal transaction object. The transaction manager also gets the timeout value of a transaction and has an internal *timeout processor* with a *timeout handler* to rollback a transaction when it expires. Such a rollback is performed via the core API as a new request.

When a user commits or rollbacks a transaction, the respective methods of the transaction manager are invoked in the corresponding task. When an implicit transaction is used, the commit method of the transaction manager is called at the end of the task, if the request processing is completed and the task will not be blocked (cf. Section 4.3 for blocking behavior). In case of an error the rollback method is called instead of the commit method. Inside these methods the wait & event manager is called (see Section 4.3.2). For tasks that use an explicit transaction the `getTransaction` method is called to get the internal transaction object for the transaction reference.

Before a transaction can be committed or rollbacked, all of its sub-transactions need to be finished, that is, committed or rollbacked. The method `lockAndWaitFor-SubTransactions` has been added to the interface `Transaction` for that purpose. Internally this method locks the transaction, that is, no new sub-transactions can be created, and then blockingly waits for the sub-transactions to finish. It is also called to prepare a transaction.

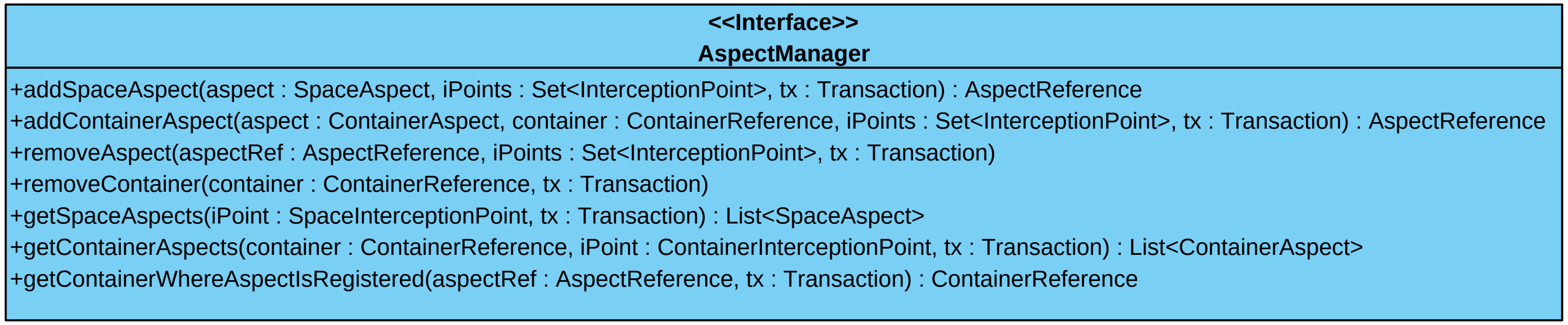


<<Interface>>
TransactionManager
+addTransaction(txRef : TransactionReference, tx : Transaction, timeout : long)
+getTransaction(txRef : TransactionReference) : Transaction
+commitTransaction(txRef : TransactionReference, implicitTx : boolean)
+rollbackTransaction(txRef : TransactionReference, implicitTx : boolean)
+shutdown()
DefaultTransactionManager
-transactions : Map<TransactionReference, Transaction>
-timedOutTransactions : Map<TransactionReference, Transaction>
-timeoutHandler : TransactionTimeoutHandler
-timeoutProcessor : TimeoutProcessor<TransactionReference>
-waitEventManager : WaitAndEventManager
-core : MzsCore

Figure 4.9: The interface and implementation of the transaction manager

4.2.5 Aspect Management and Invocation

Two components are used for aspect management in the runtime, the *aspect manager* and the *aspect invoker*. The aspect manager has a list of the aspects registered for the space or a specific container in the space. Its interface is shown in Figure 4.10. The first three methods are called from the tasks to add or remove an aspect, the fourth, `removeContainer`, is called in the `DestroyContainerTask`. The other methods are used in the aspect invoker, which is called from the request tasks to execute the registered aspects. It has the same interface as a *space aspect* (see Section 5.2). There is currently one implementation, `SerialAspectInvoker`, that executes the registered aspects in the same order as they were added to the aspect manager.



```
<<Interface>>
AspectManager
+addSpaceAspect(aspect : SpaceAspect, iPoints : Set<InterceptionPoint>, tx : Transaction) : AspectReference
+addContainerAspect(aspect : ContainerAspect, container : ContainerReference, iPoints : Set<InterceptionPoint>, tx : Transaction) : AspectReference
+removeAspect(aspectRef : AspectReference, iPoints : Set<InterceptionPoint>, tx : Transaction)
+removeContainer(container : ContainerReference, tx : Transaction)
+getSpaceAspects(iPoint : SpaceInterceptionPoint, tx : Transaction) : List<SpaceAspect>
+getContainerAspects(container : ContainerReference, iPoint : ContainerInterceptionPoint, tx : Transaction) : List<ContainerAspect>
+getContainerWhereAspectIsRegistered(aspectRef : AspectReference, tx : Transaction) : ContainerReference
```

Figure 4.10: The interface of the Aspect Manager

Aspects have a return status that is `OK`, `NOTOK`, `LOCKED`, `DELAYABLE` or `SKIP` as defined in the runtime model. The aspect invoker checks this return status and returns immediately to the task, if the status is not `OK`, as it is illustrated by Figure 4.11. In the task the execution path depends also on this status. For the status `SKIP` in a pre-aspect the execution of all following pre-aspects and the actual operation is skipped. When a post-aspect returns with `SKIP` all following post-aspects are skipped. For the other return statuses of the aspect invoker the result is handled like for the operation itself, as explained in Section 4.3. The CAPI-3 component in Figure 4.11 is just an example. The same execution sequence for the aspects is also performed for a transaction management task with the transaction manager instead of CAPI-3, but the operation itself does not return a status and cannot block. Depending on the request type, a result of a specific type is expected from the actual operation and returned in the response. When the operation is skipped by a pre-aspect and after the post-aspects no such result is available, a corresponding error is set in the response instead.

4.2.6 Container and Entry Operations

The runtime calls CAPI-3 methods to access containers and the entries in them. The main CAPI-3 interface without method parameters is shown in Figure 4.12. During the MozartSpaces startup one instance of a CAPI-3 implementation is created and then used in the tasks. Currently there are two CAPI-3 implementations, one that uses only the standard Java collections (called “javanative” or “native” implementation) and an experimental prototype based on a relational database. The details of the CAPI-3 implementations are explained in [Bar10].

4.3 Blocking Behavior

In XVSM, container and entry requests can be blocking, that is, they do not immediately return with a result because they can currently not be fulfilled. This is signaled with special return values of the CAPI-3 method calls. The result objects returned by CAPI-3 can have the status `LOCKED` or `DELAYABLE`, besides the values `OK` and `NOTOK` when the request could be processed (see Section 3.3.3). Figure 4.11 shows the possible sequences of one task execution. When the return status is `OK` the result is returned. When it is `NOTOK` an error is returned instead of a result. For the status `LOCKED` or `DELAYABLE` the task is blocked and adds itself to the wait & event manager, if the request timeout allows it.

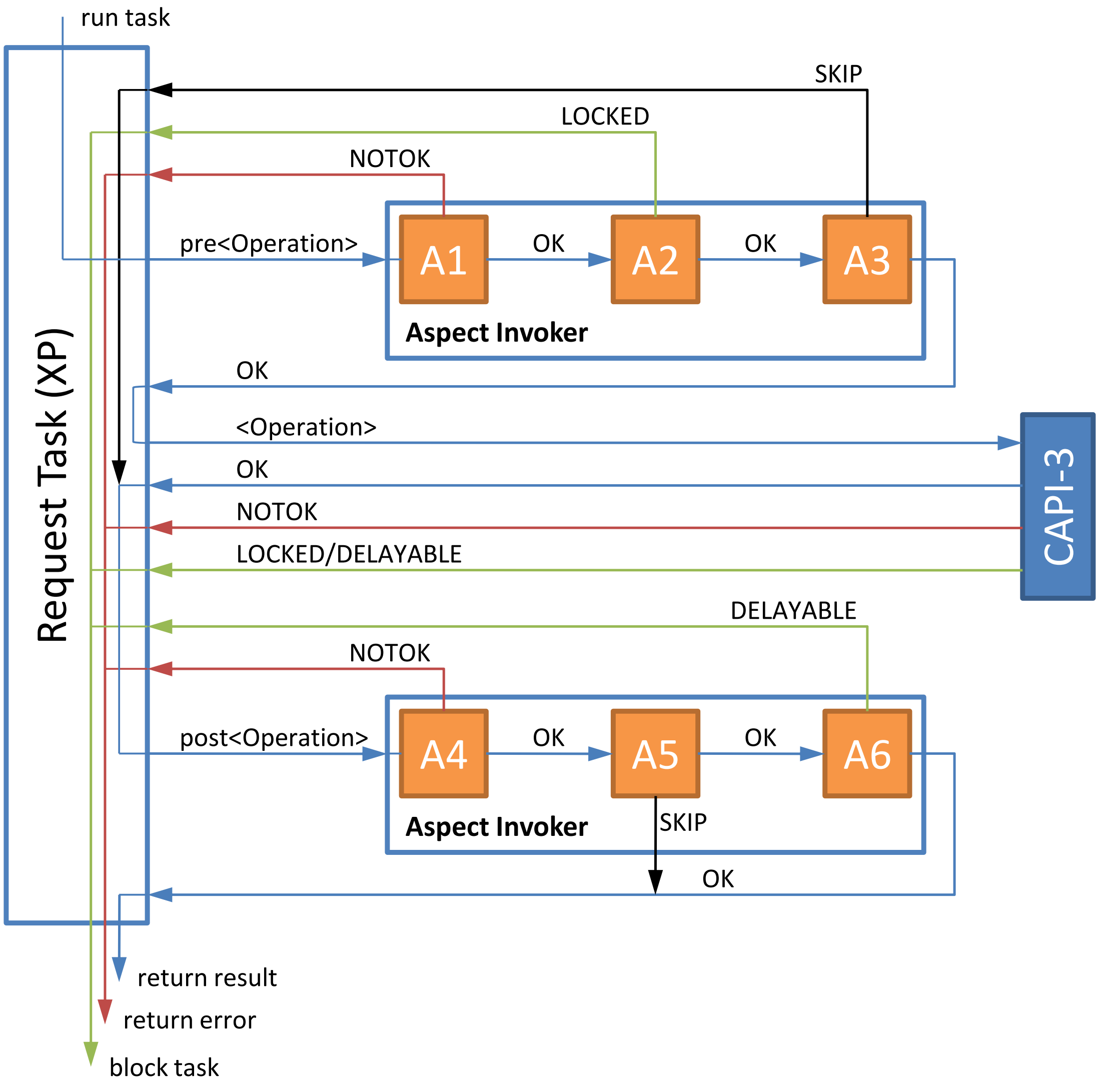
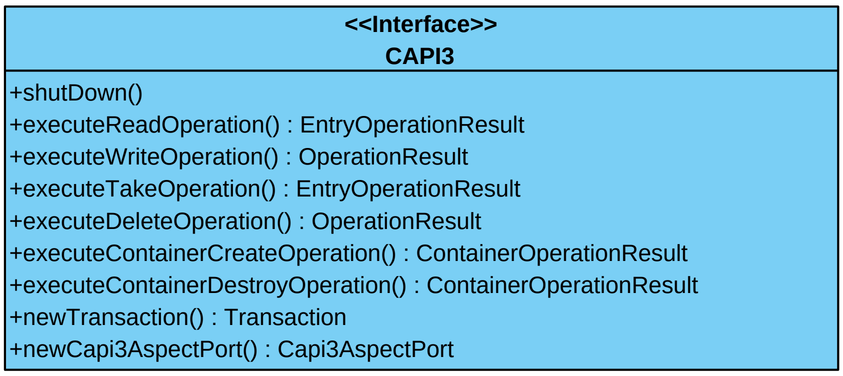


Figure 4.11: Example for a possible execution sequence of an operation and its aspects


<<Interface>>
CAPI3

+shutDown()
+executeReadOperation() : EntryOperationResult
+executeWriteOperation() : OperationResult
+executeTakeOperation() : EntryOperationResult
+executeDeleteOperation() : OperationResult
+executeContainerCreateOperation() : ContainerOperationResult
+executeContainerDestroyOperation() : ContainerOperationResult
+newTransaction() : Transaction
+newCapi3AspectPort() : Capi3AspectPort

Figure 4.12: The CAPI-3 interface [Bar10, cutout of Figure 9]

4.3.1 Timeout Processing

The blocking behavior of requests can be controlled by the user with the request timeout. Table 4.1 shows the compatibility of CAPI-3 return statuses and the timeout values. The timeout can be a positive integer value or one of the constants `INFINITE`, `TRY_ONCE` or `ZERO` explained in Section 3.3.5. If the timeout is a positive value, the task adds itself to the timeout processor after it has added itself to the wait & event manager. For an `INFINITE` timeout, or `TRY_ONCE` and the status `LOCKED`, there is no timeout in milliseconds after which the request times out. So the task adds itself to the wait & event manager but not the timeout processor. For a request timeout of `ZERO` and the status `LOCKED` or `DELAYABLE`, the task is not blocked and an exception corresponding to the reason for the `LOCKED/DELAYABLE` CAPI-3 status is sent as response, as well as for timeout `TRY_ONCE` and status `DELAYABLE`.

Table 4.1: Task blocking depending on CAPI-3 status and request timeout

	Request timeout			
Status	Integer > 0	INFINITE	TRY_ONCE	ZERO
LOCKED	block (timeout)	block	block	error
DELAYABLE	block (timeout)	block	error	error

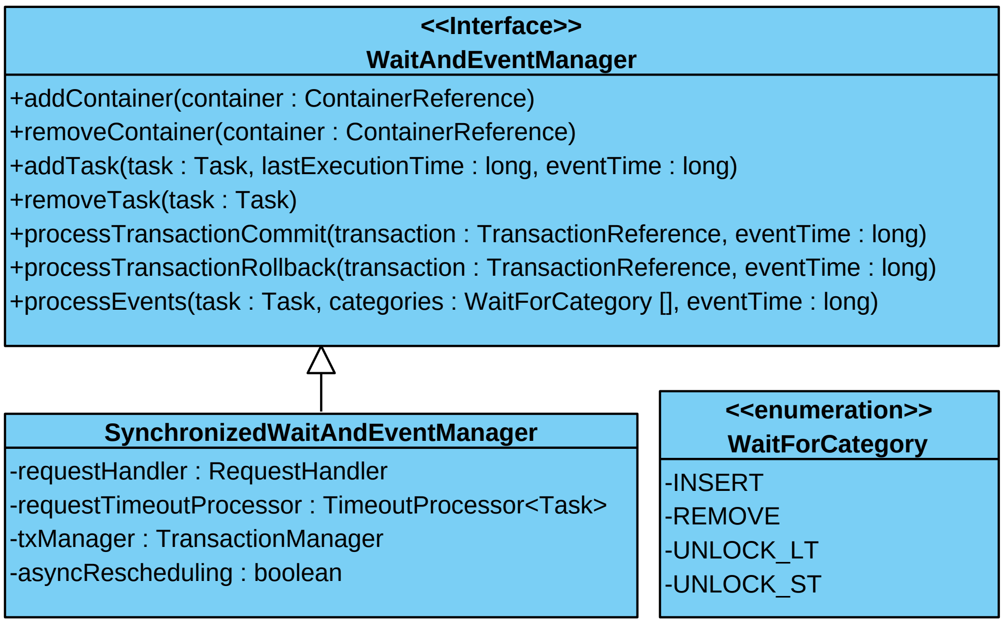
Timeout Processor

The timeout processor has a simple interface (`TimeoutProcessor`) with methods to add and remove generic elements that can time out. These generic elements have an expire time and an arbitrary object as value, for example, a task object. To react on elements that time out, a *timeout handler* can be set on the timeout processor. An instance of this timeout processor is used for the tasks; the transaction manager uses its own instance for the transactions.

The first `TimeoutProcessor` implementation used a `DelayQueue` internally. Thus, in contrast to the previous XVSM implementations, it is non-polling, that is, there is no periodic task that checks for timed out elements. However, micro-benchmarks have shown that this timeout processor is slow and a second implementation based on a `ConcurrentNavigableMap` has been implemented. There a task is scheduled periodically to check for timed out elements and passes them to the timeout handler. A non-polling variant of the second timeout processor would be possible by putting the thread that checks for timeouts to sleep, until the first timeout is elapsed or an element with an earlier timeout is added. In micro-benchmarks the second timeout processor is faster than the first one, but its integration into the runtime requires refactoring in CAPI-3 and is subject to future work.

4.3.2 Wait & Event Manager

Tasks that must be blocked add themselves to the wait & event manager, which corresponds to the wait container of the formal model. Its interface is depicted in Figure 4.13. The `addContainer` and `removeContainer` methods are called after a container has been created or destroyed. A task adds itself with `addTask` when it is blocked and is removed with `removeTask` by the timeout handler after it timed out. The methods `processTransactionCommit` and `processTransactionRollback` of the wait & event manager are called from the transaction manager when a transaction has been committed or rolled back, respectively. Furthermore, the wait & event manager processes the *events* that are generated after a task has been executed in the method `processEvents`. Internally it has a data structure with tasks and event timestamps as shown in Figure 3.6. Tasks are removed from this data structure when they time out or when matching events are processed, which is explained in the next section.



<<Interface>>
WaitAndEventManager
+addContainer(container : ContainerReference)
+removeContainer(container : ContainerReference)
+addTask(task : Task, lastExecutionTime : long, eventTime : long)
+removeTask(task : Task)
+processTransactionCommit(transaction : TransactionReference, eventTime : long)
+processTransactionRollback(transaction : TransactionReference, eventTime : long)
+processEvents(task : Task, categories : WaitForCategory [], eventTime : long)
SynchronizedWaitAndEventManager
-requestHandler : RequestHandler
-requestTimeoutProcessor : TimeoutProcessor<Task>
-txManager : TransactionManager
-asyncRescheduling : boolean
<<enumeration>>
WaitForCategory
-INSERT
-REMOVE
-UNLOCK_LT
-UNLOCK_ST

Figure 4.13: Wait & event manager types

Currently there is one wait & event manager implementation, the `Synchronized-WaitAndEventManager`. Its name comes from the complete synchronization of all its methods, that is, they are all `synchronized`. So it is relatively easy to avoid race conditions, but this reduces the possible concurrency.

4.3.3 Event Processing

In the runtime, events are generated after a request that accesses CAPI-3 has been processed. They are immediately processed by the wait & event manager. The event processing follows the formal model as described in [Cra10] and summarized in Section 3.3.5. Tasks that were added to the wait & event manager are rescheduled when matching events are processed in the method `processEvents`. Events have categories, a timestamp and are from a specific task. From the task the container and the transaction are relevant for event processing (cf. method signature in Figure 4.13). The categories for events are from the `WaitForCategory` enumeration that is used for blocked tasks. The categories where events are generated depend on the request type and its CAPI-3 return status. Additionally, events from aspects are generated when they use the special restricted CAPI-3 interface `Capi3AspectPort`. This interface is accessible from within aspects for some ipoints and provides entry operations on the container that is used in the request (see Section 5.2.2).

In the `processEvents` method, for each event category, container and transaction, the timestamps are updated and the waiting tasks for this category, on the same container and in the same transaction, are rescheduled. The events are only visible within the transaction and made generally visible when the transaction manager commits or rollbacks the transaction and calls the method `processTransaction-Commit` or `processTransactionRollback` of the wait & event manager, respectively.

When tasks are rescheduled, they are usually passed to the request handler and enqueued like new tasks. This is done synchronously in the thread that was used for processing the request. For special configurations of the request handler, where its executor service has no worker threads and executes tasks synchronously, tasks are rescheduled asynchronously in an own thread pool to avoid race conditions because of the concurrent modification of data structures in the wait & event manager. These race conditions could occur for synchronous task execution and rescheduling due to the event processing of the rescheduled task interfering with the event processing of the original task execution where it was rescheduled, because the same thread is used for both event processings and the Java monitor locks (which are behind the `synchronized` keyword) are reentrant. So, either the execution or the rescheduling of tasks has to be asynchronous.

4.3.4 Alternative Wait and Event Management

When the event processing for the formal model of XVSM and MozartSpaces 2.0 was designed, an alternative structure for the requests and events was discussed.

This data structure can be seen as an extended version of the now implemented wait & event manager and is depicted in Figure 4.14. It combines the *wait container* and the *request container* from the formal model and contains all requests in the runtime and also the event timestamps. Besides container and wait categories, also the coordinators are distinguished to allow for more efficient request rescheduling, event-request matching with finer granularity, and easier implementation of special scheduling strategies and fairness.

The request and event data structure has a simple interface with methods to put and get requests. Internally, the queue at the bottom contains references to request collections and is used to signal in which collections requests are ready to be executed. This architecture, where each request is referenced in two collections, is proposed because the otherwise needed polling or blocking read from many collections would block many threads (as many as collections where requests could be) or would be computationally expensive otherwise. The strategies for the collections define which entries (requests) are provided next. New requests are timestamped and put into the container. They are taken and executed by the XP. If a request cannot be processed, it is put back into the container. Events are processed on the requests in the container similar to the wait & event manager.

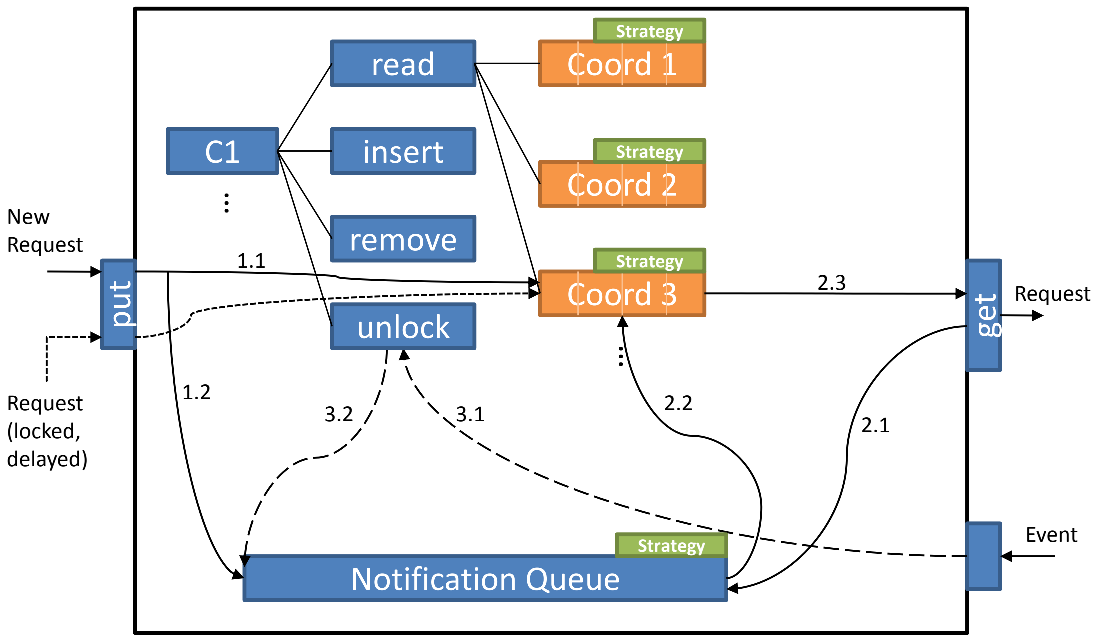


Figure 4.14: Alternative structure for requests and events

Details of Request and Event Processing

New requests are routed to the collection of the first coordinator, put there and signaled in the notification queue (see the solid lines 1.1 and 1.2 on the left in Figure 4.14). The XP takes collection references from the notification queue. It then takes a request from the referenced collection and processes it (see the solid lines 2.1, 2.2, and 2.3 on the right). Requests that are delayed because they cannot be fulfilled, are put back into the suitable request collection. Internally the request collections could be divided further, for example, there could be separate "sub-collections" for the waiting requests and the requests that should be processed. The event handling is performed on the request collections that are affected by the event, which depends on the request and the coordinators used in it. For every event a `lastEventTimestamp` in the suitable request collection is updated and the matching waiting requests are rescheduled by signaling their availability through the notification queue (see the dashed lines 3.1 and 3.2 on the left). Here the event processing logic also used in the wait & event manager, that also takes the transactions into account, could be used with small adjustments.

Strategies for Collections

As mentioned above, the collections and also the notification queue have only a simple interface to put and get requests, the actual behavior is determined by a queuing strategy. There could be such a strategy for the notification queue to provide the collection references in a round-robin manner, prioritize certain containers, or order them by various criteria. Strategies for the request collections could be coordinator specific with a simple queue as default implementation.

If the coordinator provides a matching strategy, this could be used for a more accurate matching of waiting requests and events. As the coordinator also controls the information that is returned with the request status, e.g., the entry count and a key, this information could be used for event processing in addition to generic information like the container reference and event category. For some coordinators, new requests do not need to be executed, if "similar" requests are already waiting. They can be put directly to the other waiting requests. An example for this is a read request for one entry with the random coordinator. For other coordinators, e.g., the key coordinator, this logic is different.

Comparison with Separated Request and Wait Containers

The request and event management described in this section is clearly more complex than the variant from the section before and the question is, whether the advantages are worth the effort and outweigh the disadvantages. This also depends on the strategies for the collections, which could be provided by the coordinators and put requests, that would be blocked, directly to the waiting requests.

The advantages of this proposal are the following:

1. The performance is better when requests can be delayed immediately and do not need to be processed. This could be significant if the CAPI-3 implementation or aspects are slow. By the way, not executing the aspects here would change the semantics of the aspect execution.

2. The implementation of accurate, coordinator dependent matching of events and waiting requests is easier and more flexible. Avoiding unnecessary request rescheduling improves the performance.

3. The behavior of the runtime regarding correctness and fairness can be changed easier because the logic is in one place.

The disadvantages of this proposal are the following:

1. The complexity of the data structures is bigger because the coordinators are also distinguished and there are many request queues.

2. There is an overhead for the notification queue. An additional operation for every request is required when it is inserted into or taken from the container.

3. The event processing is slower because several coordinator collections can be affected by one event. This is a result of the deeper structuring.

4. When coordinator-specific matching strategies are used, this further slows down the event processing. Additionally, coordinator logic is executed twice when a request is woken up, in the event processing and in the actual request processing.

5. The clear separation of the runtime and the coordinators in the CAPI-3 layer is lost, if coordinator-dependent logic is used in the runtime.

The alternative wait and event management has not been implemented because the overhead of the more complex event handling would probably exceed the effect of avoiding unnecessary request processing. We expect that most requests can be directly processed without blocking and only few requests are in the wait container. So the advantage would be very little but the slower event processing would affect all requests. Particularly, we want to keep the “normal case”, that is, a request that can be processed with the first execution and does not wake up other requests, fast.

4.3.5 Deadlock Handling

In general, deadlocks occur when a circular wait situation exists. For example, task A locks resource R1 and tries to acquire the lock for resource R2 while task B locks R2 and tries to acquire the lock for R1. Additionally the following deadlock conditions must be fulfilled [CES71]:

- Mutual exclusion: tasks can exclusively lock a resource, so that no other request can access it.
- Hold and wait: a task can hold locks and blockingly wait for resources that are locked by other tasks.
- No preemption: tasks cannot be forced to release locks, they can only be released by the task that holds the lock.

Deadlocks can occur in XVSM between two or more transaction whose requests are processed in interleaving order, which is illustrated by the following example.

Example Deadlock

A minimal example of a deadlock is described in [Kar09] and shown in Figure 4.15. It comprises two transactions, tx1 and tx2, each consisting of a take and a read request. In tx1 the request R1t takes entries from container A and the request R1r reads from container B. In tx2 the request R2t takes entries from container B and the request R2r reads from container A. If the requests access the “same data” in the respective containers, a deadlock can occur as follows. First R1t and R2t are processed in any order, followed by R1r and R2r, again in any order. When the take requests are processed, an exclusive lock is set[1] and they return with status `OK`. If now the read requests are executed, they return with status `LOCKED`, because the locks set in the take operations are still present. Those locks are removed only when the transactions commit or roll back. But this is not going to happen because the read requests, that are part of the transactions, are both waiting for the release of the locks.

[1]For the sake of simplicity we can assume that the whole container is locked, as depicted in Figure 4.15. Which data is actually locked depends on the respective coordinator and the isolation level.

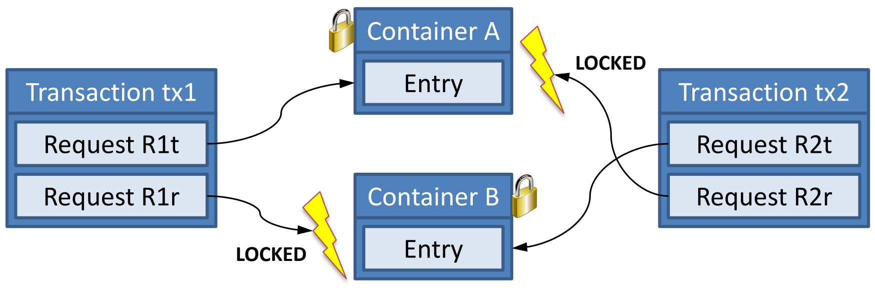


Figure 4.15: Deadlock example

Deadlock Detection

In the previous XVSM implementations deadlocks are neither prevented nor detected but resolved by transaction timeouts [Kar09]. It would be to restricting and costly to prevent or avoid them. But it is relatively easy to detect and resolve such deadlocks. This is also the approach used for deadlock handling in many systems that use pessimistic locking as a form of concurrency control, for example databases. For tuple spaces, we have found no information about deadlock detection in the literature and the JavaSpaces implementations Outrigger, GigaSpaces XAP, and Blitz. But in JavaSpaces, Jini transactions with a user-specified lease time are used, and the transaction manager aborts transactions when the lease expires. This is very similar to the behavior in the previous XVSM implementations where a transaction timeout is supported.

For MozartSpaces 2.0 we use a graph-based approach to detect deadlocks, like most other systems with deadlock handling do.

Transaction Wait-for Graph

We can depict the transactions in the XVSM core with a wait-for graph (WFG). A WFG is a directed graph (digraph) with distinct arcs (directed edges) and no loops, similar to a state graph in [CES71]. Here the vertices correspond to the transactions and the arcs show a “wait-for” relation because of a lock. A transaction X within which data is accessed that is exclusively locked by another transaction Y, is shown as the vertex X with an arc to the vertex Y. Figure 4.16 shows the WFG for the deadlock example that was described above. We can see that there is a cycle in the WFG, the necessary and sufficient condition for a deadlock.

For creating the WFG we need to consider only requests that return with status `LOCKED(lt)`, that is, data that is needed to process the request in CAPI-3 is locked by another transaction where the sub-transaction is already committed (thus `lt` for “long-term” lock). Requests that return with status `LOCKED(st)` are rescheduled

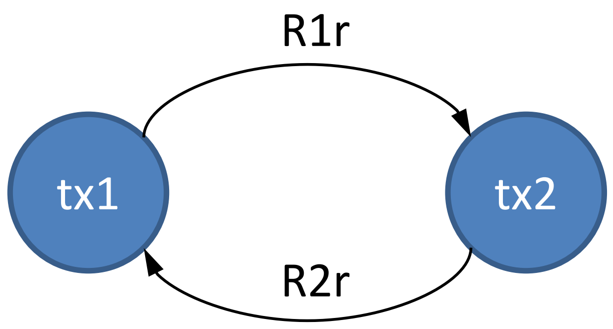


Figure 4.16: Wait-for-graph for the example deadlock in Figure 4.15

after the request that holds the corresponding short-term lock has been processed. Because the request processing in CAPI-3 is carried out non-blockingly, no deadlock can occur in that case. Requests that return with status `DELAYABLE` can be woken up by new requests, hence they cannot cause deadlocks. Requests that return with status `OK` or `NOTOK` are completed and never put into the wait container.

Whenever a request returns with status `LOCKED(lt)`, an arc from the transaction of this request to the transaction of the request that set the exclusive lock is added to the WFG[2]. Whenever a locked request is rescheduled, the arc is removed from the WFG[3]. The arcs in the WFG are distinct, that is, only one arc between two vertices is allowed. This makes it easier to detect deadlocks. A list of the locked requests is stored for each arc. So, if an arc already exists between two transactions, the request is only added to that list. This can only happen if requests of the same transaction are asynchronously sent to the runtime. Similarly, if a request from an arc with more than one waiting requests is rescheduled, only the request is removed from the list and not the whole arc.

Deadlock Detection Triggering and Algorithm

The deadlock detection can be triggered whenever an edge has been added to the wait-for graph, that is, each time a request returns with status `LOCKED(lt)`. Then only one deadlock can exist, which is immediately detected. However, depending on the number of locked requests and the cost of the deadlock detection, this may impair the overall performance of the runtime. Therefore, the deadlock detection could be started periodically. But this would be unnecessary polling, if no request had been locked in the last period. Instead, the deadlock detection should be triggered once for a period in which at least one request got locked. A timer is started when a

[2]The vertex for a transaction is implicitly added to the WFG if it is not present.
[3]The vertex for a transaction is implicitly removed from the WFG if its degree is 0.

request is locked. When the timer expires, the deadlock detection is started. If more than one request got locked, more than one deadlock could be present, at most as many as requests got locked.

The deadlock detection itself can be performed with algorithms that find cycles in a digraph. In [Hol72] an algorithm that "reduces" the WFG by successively marking (removing) sinks, vertices without outgoing arcs, is presented. If the WFG cannot be completely reduced, at least one deadlock exists and the remaining vertices are involved. The time complexity of this algorithm is $\mathcal{O}(|V| + |A|)$, where $|V|$ is the number of vertices and $|A|$ the number of arcs. Tarjan [Tar72] invented an algorithm to find the strongly connected components (SCCs) of a digraph that is based on depth-first search (DFS). A digraph is strongly connected if there is a path from each vertex to every other vertex. The SCCs of a digraph are its strongly connected subgraphs and if they all contain only one vertex (trivial SCCs) the digraph is acyclic. Like for DFS, the time complexity for Tarjan's SCC algorithm is $\mathcal{O}(|V|+|A|)$ and it can be adapted easily to return the nontrivial SCCs of a digraph. The vertices in such a nontrivial SCC correspond to the transactions that are involved in one deadlock. There are other algorithms that find the SCCs of a digraph also in $\mathcal{O}(|V| + |A|)$, but they are either slower by a constant factor (Kosaraju's algorithm) or very similar but not so widespread (Gabow's algorithm)[Sed03].

As mentioned above, many other systems with deadlock detection use a graph-based approach, usually WFGs. [GR93] describes their use for transaction processing, [Cou05] for distributed transactions and [CB05] for (distributed) database systems. In practice, most current database systems support deadlock detection for transactions. For example, Apache Derby [22], an open source relational database management system implemented in Java, detects deadlocks by searching the graph of all locks for cycles with a variant of DFS. It aborts the transaction which holds the fewest number of locks and provides an error with a detailed message containing the involved transactions, SQL statements, and locks. Apart from database systems, a DFS-based deadlock detection algorithm is also used in the Sun Java HotSpot Virtual Machine to detect thread deadlocks. [GR93] lists the C source code for deadlock detection for transaction processing systems, also based on DFS.

Deadlock Resolution

When a deadlock exists, it can be resolved by rolling back, one or more transactions that are part of it. When the WFG is tested for deadlocks after each locked request, only one deadlock can exist and only one transaction needs to be aborted to resolve the deadlock. Otherwise, when the deadlock detection is performed less frequently, it may be necessary to rollback several transactions. We can select one transaction, rollback it and check for deadlocks again or try to find the minimal set of transactions that need to be aborted to resolve the deadlock. The latter is an NP-complete problem, but in [TC99] several heuristics are presented, which select a transaction based on the WFG structure. The heuristic that performed best in their simulations selects the transaction where the maximal number of transactions can be reduced, using the reduce algorithm mentioned above. This algorithm is in $\mathcal{O}(|V|(|V| + |A|))$ time, as the reduce algorithm is performed for each vertex. It is optimal for WFGs where each vertex can have one outgoing arc at most, which is the case for synchronously used XVSM transactions.

For heuristics like the one mentioned above, it is assumed that aborting the minimal set of transactions is “optimal”. This might not be the case and the cost for selecting a transaction to abort needs to be considered as well. Several other heuristics could be used for that:

1. The transaction the most transactions are waiting for is selected. This transaction corresponds to the vertex with the highest indegree in the WFG.
2. The transaction with the fewest operations in the transaction log is selected. This is an attempt to select the transaction with the minimal cost for rollback.
3. The transaction with the last processed locked request on an outgoing arc is selected. This aborts the transaction with the request that closed the cycle.
4. The transaction with the lowest or highest start timestamp is selected.
5. No transaction is selected in the Runtime, but a special error is reported to the users of the transactions that are part of the deadlock. If none of the transactions is rolled back then, the deadlock is not resolved.

In any case, a detailed error message with all involved transactions and locks should be returned to the user.

Implemented Basic Variant

Because deadlocks are probably rare and occur just temporarily when transaction timeouts are used, only a basic variant of the deadlock handling described above is included in the first release of MozartSpaces 2.0: every time a task is added to the WFG it is searched for deadlocks with Tarjan’s SCC algorithm. If a deadlock is found, this is only logged and no deadlock resolution is performed. Furthermore, the deadlock detection is deactivated by default and needs to be activated manually.

4.4 Startup and Configuration

To access a space, a MozartSpaces instance needs to be created. This is done by a factory method using the passed configuration. The configuration parameters influence mainly the remote communication and the threading. The configuration is currently static and cannot be changed after the core has been created. The configuration is passed to the factory via configuration objects. These configuration objects can be read from a configuration file or created programmatically. A user creates a MozartSpaces instance with the API that wraps the factory and reads the configuration file. This is explained in Section 6.3.

The important configuration for the runtime is the XP thread pool, the entry copier and whether a core has a runtime at all. As explained in Section 3.1, a MozartSpaces core can be a client with no embedded space and hence no request handler, XP or other runtime components.

4.4.1 Entry Copier

The entry copier is a small optional component that copies entries and if configured also the request context. This is useful to avoid that the objects in the space can be directly changed because of direct references from the application to an embedded MozartSpaces core. Only when a request or response is transferred to another core, the entries are copied automatically because the message is serialized. To use a serializer for copying entries is one option, the other one is to implement an interface with a method to clone the object for every class that is stored in the space.

4.4.2 XP Thread Pool Configuration

The configuration of the XP thread pool determines the concurrency behavior of the runtime. The thread pool is an instance of the interface `ExecutorService`. Currently three implementations of it can be configured:

1. The executor service is a thread pool with a *fixed number of worker threads*. It is created with a factory method in the class `Executors` that is included in the Java API and the number of threads can be configured. The threads execute the tasks and are reused.

2. The executor service is a thread pool with an *unbounded number of worker threads*. It is also created with a factory method in the class `Executors`. New threads are created when tasks are added to the request handler and no thread

is free. The threads are reused and threads that are unused for 60 seconds are terminated.

3. The executor service is an instance of `WithinThreadExecutorService`, which directly executes tasks that are added on the same thread. There are *no worker threads* and the tasks are executed synchronously, in contrast to the asynchronous execution when a real thread pool is used. Because multiple threads of the application (for embedded requests) or the receiver may be used, concurrency is still possible with this configuration.

The XP thread pool configuration affects the rescheduling of blocked tasks. If the `WithinThreadExecutorService` is used, rescheduled tasks are executed with an additional single thread executor service and not the main executor service of the request handler (see Section 4.3.3).

The different thread pools mentioned above have different performance and scalability characteristics. The variant without worker threads is the fastest because there is no thread context switch necessary (see benchmark results in Chapter 8). However, because the thread that adds the request to the request handler is reused, it is neither suitable for use with the asynchronous core API (see Section 6.1) nor for a transport protocol implementation without a thread pool nor when threads should not be blocked at all. The thread pool with a fixed number of worker threads can be used in such cases. It decouples adding a request to the request handler and processing it. The number of threads can be configured in a configuration file (see Section 6.3). There is also the option to use a thread pool with an unbounded number of worker threads instead of a fixed (maximal) number. This option should be used with care, especially together with asynchronous core API calls. A thread is relatively heavyweight and uses system resources. If the number of running threads is too large this could cause an application crash, e.g., because the JVM is out of memory. The exact limit for the number of threads depends on the JVM, its configuration and the underlying operating system.

5 Extensibility, Remoting and Interoperability

MozartSpaces is designed to be extensible in several ways. The coordination can be defined with custom coordinators which is explained in Section 5.1. Aspects can be used to execute arbitrary code before or after a request is processed in the runtime. They are used to implement notifications and are described in Section 5.2. Section 5.3 illustrates the remote communication in MozartSpaces, how messages are sent and received and how the transport protocols are selected and can be replaced by custom implementations. The wire protocol of the remote communication in XVSM is defined with the XVSMP, which is described in an interoperable XML-based form in Section 5.4.

5.1 Custom Coordinators

In XVSM, entries in containers are managed by coordinators that are specified when the container is created. A container's coordinators can be one or more of the predefined coordinators (see Section 3.3.4) and/or custom coordinators. Every coordinator and the corresponding selector have to implement special interfaces. In the Core API, a coordinator has to implement the interface `Coordinator` and a selector the interface `Selector`. This coordinator and selector instances are also passed to CAPI-3, where internally other types and classes can implement the actual coordination logic. In the CAPI-3 Java Native implementation, coordinators have to implement the interface `NativeCoordinator`, which extends `Coordinator`, and selectors the interface `NativeSelector`, which extends `Selector`. The coordinator-specific `CoordinationData` that is used to write entries, can be directly used in CAPI-3.

The details of CAPI-3 Java Native and the interfaces for coordinators and selectors are explained in [Bar10]. Important for the extensibility is that there are *generic* coordinators and selectors, that are independent of the concrete CAPI-3 implementation and are used at the API level. They need to be translated to the

CAPI-3 specific internal implementations. For example, the generic predefined FIFO coordinator is represented by the class `FifoCoordinator` at the core API level and implemented in the class `DefaultFifoCoordinator` in CAPI-3 Java Native. The translation between generic coordinator objects and internal objects is performed inside CAPI-3 Java Native and included for all predefined coordinators. But for custom coordinators the translation needs to be extended. An easy way is to create a map of *coordinator translators* that can be configured from the outside and consists of coordinator translators that can translate a specific generic coordinator to the internal coordinator implementation. Currently this map does not exist and the source code of CAPI-3 would have to be changed for every new custom coordinator. Of course this is not practicable and the workaround is to use the internal coordinator implementation also in the API, which is possible because `NativeCoordinator` extends `Coordinator`. The implementation of the above-mentioned map for the coordinator translators is future work.

In addition to the translation between coordination classes at the API level and inside CAPI-3, the creation of coordination classes is necessary for the interoperable remote communication. The information about the coordinators, selectors and/or the coordination data, together with their properties, is serialized as part of request messages (see the XML wire protocol in Section 5.4 as example). For the deserialization a *creator*, that creates the coordination classes corresponding to the coordination information in the serialized request, is required for each coordinator. For the predefined coordinators such creators are implemented, but fur custom coordinators they need to be written manually. The API method to add such creators is subject of future work.

5.2 Aspects

Aspects can be used to execute arbitrary code before or after a request is processed in the runtime. When a request is passed to the request handler, a task is created to process this request. This task calls the aspects before and after the actual request is processed. More precisely, the task calls the *aspect invoker* which gets the aspects from the *aspect manager* and executes them (see Section 4.2.5). Aspects can be added to and removed from the aspect manager for specific interception points (ipoints). Every ipoint corresponds to a different aspect method. Figure 5.1 shows the ipoint enumerations and Figure 5.2 the two corresponding aspect interfaces with the aspect methods. As you can see from these figures, we differentiate between *container aspects* and *space aspects*.

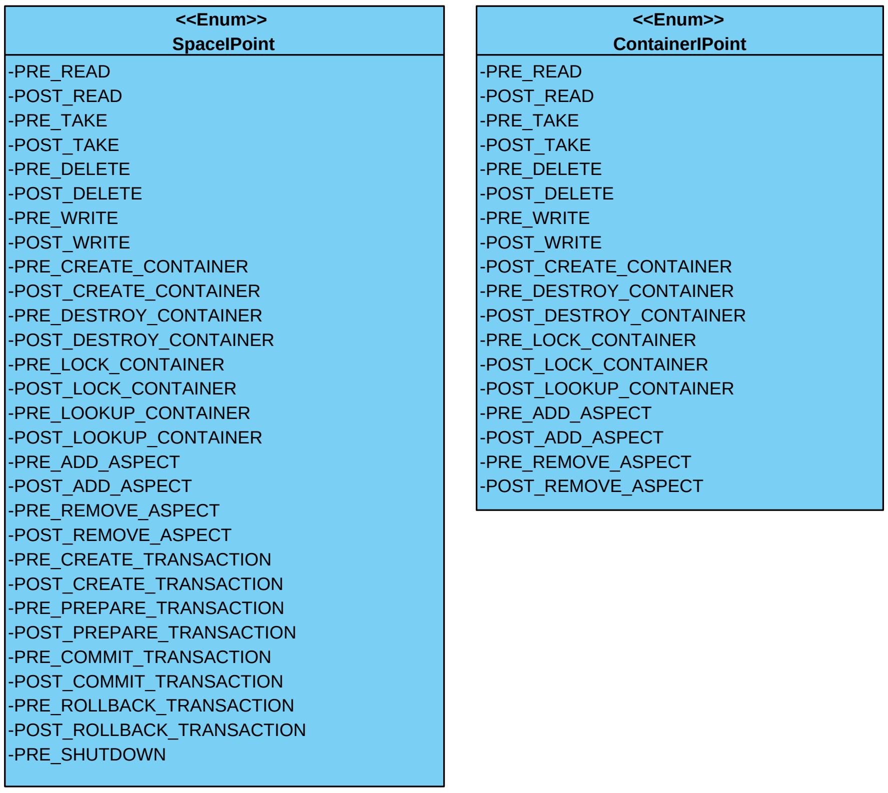
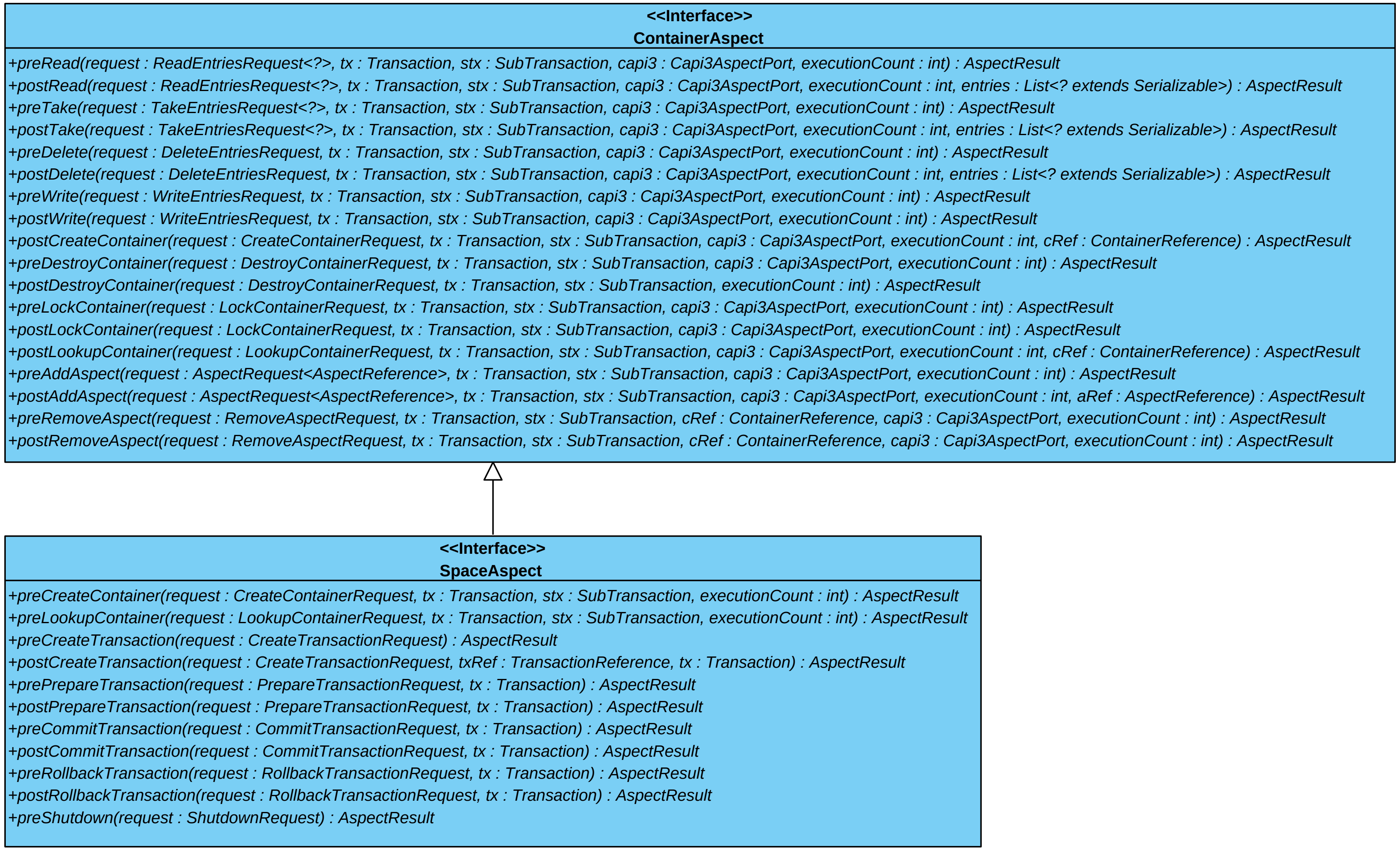


Figure 5.1: Interception points for aspects



```
<<Interface>>
ContainerAspect
+preRead(request : ReadEntriesRequest<?>, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int) : AspectResult
+postRead(request : ReadEntriesRequest<?>, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int, entries : List<? extends Serializable>) : AspectResult
+preTake(request : TakeEntriesRequest<?>, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int) : AspectResult
+postTake(request : TakeEntriesRequest<?>, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int, entries : List<? extends Serializable>) : AspectResult
+preDelete(request : DeleteEntriesRequest, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int) : AspectResult
+postDelete(request : DeleteEntriesRequest, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int, entries : List<? extends Serializable>) : AspectResult
+preWrite(request : WriteEntriesRequest, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int) : AspectResult
+postWrite(request : WriteEntriesRequest, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int) : AspectResult
+postCreateContainer(request : CreateContainerRequest, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int, cRef : ContainerReference) : AspectResult
+preDestroyContainer(request : DestroyContainerRequest, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int) : AspectResult
+postDestroyContainer(request : DestroyContainerRequest, tx : Transaction, stx : SubTransaction, executionCount : int) : AspectResult
+preLockContainer(request : LockContainerRequest, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int) : AspectResult
+postLockContainer(request : LockContainerRequest, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int) : AspectResult
+postLookupContainer(request : LookupContainerRequest, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int, cRef : ContainerReference) : AspectResult
+preAddAspect(request : AspectRequest<AspectReference>, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int) : AspectResult
+postAddAspect(request : AspectRequest<AspectReference>, tx : Transaction, stx : SubTransaction, capi3 : Capi3AspectPort, executionCount : int, aRef : AspectReference) : AspectResult
+preRemoveAspect(request : RemoveAspectRequest, tx : Transaction, stx : SubTransaction, cRef : ContainerReference, capi3 : Capi3AspectPort, executionCount : int) : AspectResult
+postRemoveAspect(request : RemoveAspectRequest, tx : Transaction, stx : SubTransaction, cRef : ContainerReference, capi3 : Capi3AspectPort, executionCount : int) : AspectResult

<<Interface>>
SpaceAspect
+preCreateContainer(request : CreateContainerRequest, tx : Transaction, stx : SubTransaction, executionCount : int) : AspectResult
+preLookupContainer(request : LookupContainerRequest, tx : Transaction, stx : SubTransaction, executionCount : int) : AspectResult
+preCreateTransaction(request : CreateTransactionRequest) : AspectResult
+postCreateTransaction(request : CreateTransactionRequest, txRef : TransactionReference, tx : Transaction) : AspectResult
+prePrepareTransaction(request : PrepareTransactionRequest, tx : Transaction) : AspectResult
+postPrepareTransaction(request : PrepareTransactionRequest, tx : Transaction) : AspectResult
+preCommitTransaction(request : CommitTransactionRequest, tx : Transaction) : AspectResult
+postCommitTransaction(request : CommitTransactionRequest, tx : Transaction) : AspectResult
+preRollbackTransaction(request : RollbackTransactionRequest, tx : Transaction) : AspectResult
+postRollbackTransaction(request : RollbackTransactionRequest, tx : Transaction) : AspectResult
+preShutdown(request : ShutdownRequest) : AspectResult
```

Figure 5.2: Interfaces with aspect methods

5.2.1 Container and Space Aspects

Container aspects (or local aspects) can be added for operations on a specific container, that is, when a container reference is a property in the request or the result of the request. This is obvious for entry requests, because they always specify a specific container, and for the requests to lock or destroy a container. When a container is created or looked up, the container reference is available after the operation and container aspects can be added there. For the aspect management requests the container reference is optional, the container aspects are only executed when one is specified, that is, an aspect is added or removed for a specific container.

Space aspects (or global aspects) are always invoked before or after an operation on the space. They can be added for interception points of space operations without a container reference, for example transaction management requests, and for the interception points where container aspects can be added, where they are executed for every request, regardless of the container reference. Additionally, every space aspect can also be used as container aspect, that is, added for a specific container and container ipoints, because the interface `SpaceAspect` extends the interface `ContainerAspect`. Therefore, the space ipoints are a superset of the container ipoints. The reason we use two separate enumerations for the ipoints is that Java does not allow to extend enumerations.

5.2.2 Aspect Methods

The interfaces with the method signatures that aspects have to implement are listed in Figure 5.2. There is a method for each ipoint before (pre) and after (post) a request is processed. The only exception is the shutdown, where only a pre-shutdown ipoint (and corresponding method) exists. For example, when a `ReadEntriesRequest` is processed in a `ReadEntriesTask`, the `preRead` method of the aspects is called before the actual read operation is performed in CAPI-3, and the `postRead` method afterwards.

The aspect methods have not only different names for the ipoints but also differ in the parameter list. Every method has the request as parameter, so aspects can access all request properties. In particular, the request context can be accessed and properties can be read and written there. This context object can be used to pass arbitrary objects from API calls to aspects. Apart from the request, all “post” methods get the result of the operation. This result is equal to the result of the request, except for the deletion of entries where the deleted entries are passed to the aspects but not returned as request result. And, for the transaction creation the internal transaction object and the transaction reference are passed to the aspects, while only the reference is returned as result. Furthermore, aspect methods for transactional requests get the internal transaction and sub-transaction objects as well as the *execution count*, a counter variable with the repetition number the task for the request is currently executed. For methods of requests that have a container reference property, a special CAPI-3 interface `Capi3AspectPort`, that is limited to entry operations on the container specified by the container reference, is provided. This interface allows to directly call operations that otherwise would require a new request that is processed by the runtime. This can be useful to make changes to a container that would otherwise result in an infinite loop. For example, a pre- or post-take aspect that also takes entries from the container would trigger itself infinite recursively with a new core API request, but can now use the `Capi3AspectPort` for its own take operation.

All aspect methods have a return value of the type `AspectResult`. This class encapsulates the result status of the aspect (one of the values of the enumeration `AspectStatus`—`OK`, `NOTOK`, `LOCKED`, `DELAYABLE` or `SKIP`) and, optionally, an exception. This exception should be the error for the status `NOTOK`, or the reason for the status `LOCKED` or `DELAYABLE` as defined in CAPI-3 (see [Bar10]). The handling of the aspect status is described in Section 4.2.5. The aspect methods cannot return a value that is used as result for the request. They can modify the request context and the entry list, when entries are written or after they have been selected (read, take, delete), but cannot directly change the result otherwise.

5.2.3 Aspect Manager

The aspect manager is described from the runtime perspective in Section 4.2.5. It stores for every aspect that is added, for which ipoints and for container aspects also for which container it is registered. This information is used by the aspect invoker when the aspects are executed. The aspect manager creates and returns an aspect reference when an aspect is added. This aspect reference can be used later to remove the aspect from all or only a part of the registered ipoints. The current aspect manager implementation, `SimpleAspectManager`, is not transactional and does not support isolation, that is, it ignores the passed transaction.

5.2.4 Aspect Invoker

The aspect invoker calls aspects that have been added previously. It has the same interface as a space aspect and its aspect methods are called from the respective request tasks. The current implementation, the `SerialAspectInvoker`, gets the registered aspects from the aspect manager and invokes them in the same order as they were added to the aspect manager. When container and space aspects can be registered, for “pre”-ipoints first the space aspects and then the container aspects are invoked, and for “post”-ipoints first the container aspects and then the space aspects are invoked.

We can illustrate the execution sequence with an example, where two aspects, a space aspect sa1 and a container aspect ca1, are both added for the ipoints `PRE_READ` and `POST_READ`, ca1 specifically for the container c1. When now entries are read from c1, the aspect invoker’s `preRead` method is called from the `ReadEntriesTask`. It invokes the correspondent methods of sa1 and then ca1. Back in the task, the read operation of CAPI-3 is called and then the `postRead` method of the aspect invoker, which invokes the correspondent methods of ca1 and then sa1.

5.2.5 Notifications

Notifications in MozartSpaces allow a user to get informed when the contents of a container is accessed or changed. A user can create a notification for a container and an operation and specify a listener that is called each time when an operation has been performed on the container.

Notifications are implemented as an extension of the basic space functionality with the help of aspects and a small API that allows for creating and destroying notifications. We have implemented one specific notification flavour that is similar to the notifications in MozartSpaces 1.0 and the *Notification1* definition and its implementation in XcoSpaces [Sch08a].

The user can create a notification by using the `createNotification` method of the `NotificationManager`. The notification is represented by a `Notification` object that can be used to remove the notification. When a notification is created, the reference of a container, a listener object with a callback method and one or more operations (`WRITE`, `READ`, `TAKE`, `DELETE`) can be specified. Internally, a *notification container* is created on the core of the container that should be observed. A *notification aspect* for the “post”-ipoints of the specified operations is added to the container that should be observed. Then the notification object is created and an internal thread to take entries from the notification container, the *notifier*, is created.

Figure 5.3 shows the basic structure of the notification. The notifier tries to take entries from the notification container in a loop, until the notification is removed. The take request is blocked when no entries are available. When an application performs an operation on the application container that is observed, for example, it deletes entries from the container, the notification aspect writes a *notification entry* to the notification container. This notification entry contains the operation type, `DELETE` in our example, and the entries affected by the operation, the deleted entries in our example. The notification entry is taken by the notifier, which calls the observer, that is, the callback method of the listener object that has been specified when the notification has been created.

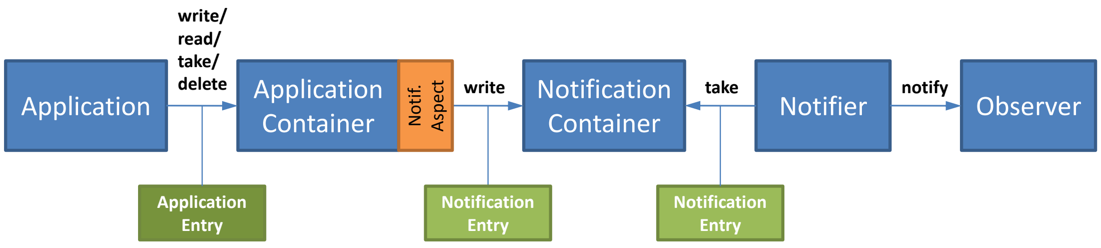


Figure 5.3: Structure of the notification implementation with aspects

By default, the notifier and the notification aspect use implicit transactions. Optionally, a transaction can be specified when the notification is created. This transaction is then used to set-up the notification and the notifier uses it to take entries from the notification container. The notification aspect uses the transaction of the request on the application container to write the notification entries into the notification container. If no transaction is specified for the notification or the application uses another transaction than the notification, the notifier and the notification aspects use different transactions and the observer will be notified about all operations when the transaction used by the application is committed. However, when the notification and the application use the same transaction, the notification entries in the notification container are immediately visible for the notifier and thus the observer is notified directly after each operation.

The current notification implementation is very simple, extending the functionality and improving the implementation itself is future work. For example, notifications could be more selective, that is, notify an observer only when entries that match a specific selector are affected by an operation.

5.3 Remote Communication

Remote communication is necessary in MozartSpaces to access the space of other cores on the same or a remote computer. Request and response messages need to be sent and received. In the formal model and also in the overall runtime structure (see Figure 4.1) there is a *sender* for sending messages and a *receiver* for receiving messages. In the implementation more than one sender and receiver is supported. There is a sender and receiver pair for each transport protocol and a *communication manager* that selects the sender for a request. The implementation of one transport protocol, TCP with sockets, is included in the runtime. Figure 5.4 shows the important interfaces and classes for transport protocols, the communication manager and the TCP socket transport. Additional transport protocols can be added to the communication manager, but changes to the runtime factory (see Section 4.4) are necessary, as there is currently no API support for that.

5.3.1 Communication Manager

When the *communication manager* is constructed it gets the senders and receivers. For the senders this is a map where the scheme for a sender is used as key. The current implementation `SimpleCommunicationManager` implements the interface `Sender`, in its method `sendMessage` the sender is selected based on the scheme of the destination space URI of the message. This value can be specified by the user in the API (see Section 6.1). When, for example, a request is sent to the URI `xvsm://localhost:9876`, the communication manager selects the sender for the scheme `xvsm` and passes the message to it. There is also a default scheme to handle URIs without a scheme, for example `//localhost:9876`.

Besides the selection of the sender, the `SimpleCommunicationManager` calls the shutdown methods of the senders and receivers, when it is called to shutdown itself. It will be extended as part of future work as outlined in Chapter 9.

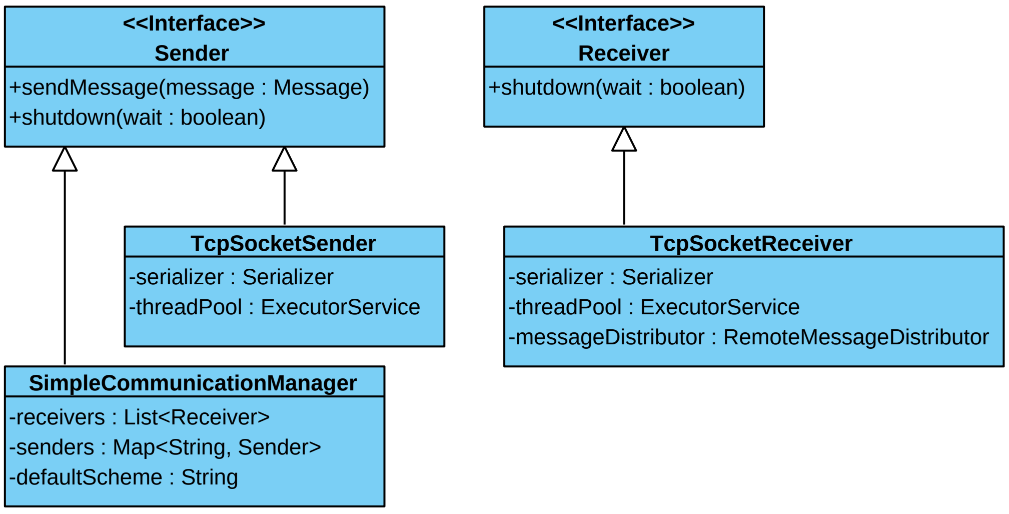

<<Interface>>
Sender
+sendMessage(message : Message)
+shutdown(wait : boolean)
<<Interface>>
Receiver
+shutdown(wait : boolean)
TcpSocketSender
-serializer : Serializer
-threadPool : ExecutorService
TcpSocketReceiver
-serializer : Serializer
-threadPool : ExecutorService
-messageDistributor : RemoteMessageDistributor
SimpleCommunicationManager
-receivers : List<Receiver>
-senders : Map<String, Sender>
-defaultScheme : String

Figure 5.4: Classes for the remote communication

In contrast to the senders, where the communication manager selects the appropiate sender, the receivers are currently independent of the communication manager for transporting messages. A receiver has usually a listener that waits for incoming connections and receives messages from such incoming connections. After deserializing a message the receiver passes it to the `RemoteMessageDistributor` (cf. *Message Distributor* in Figure 4.1). While the senders and receivers are currently independent classes, the sender and receiver for a specific transport protocol of course need to be compatible.

When a sender is called from the runtime via the communication manager, the message can be buffered and does not need to be sent immediately in the same thread. If messages are sent asynchronously, they must be serialized before the method `sendMessage` returns, because otherwise objects referenced in messages could be changed after the message is created but before it is sent.

5.3.2 TCP Socket Transport

The TCP socket transport is implemented directly with the socket and I/O classes of the Java API. The sender is implemented in the class `TcpSocketSender` and the receiver in `TcpSocketReceiver`, as shown in Figure 5.4. We decided to use the "old" blocking I/O classes (package `java.io`) and not the newer non-blocking I/O classes (package `java.nio`, called New I/O or short NIO), because they are easier to use and we wanted to start with a basic remote communication implementation. For the future, the TCP socket transport implementation can be replaced by a faster and more scalable implementation that uses NIO. Especially applications with thousands of open connections would benefit from the use of NIO, but we expect to have less open connections for most use cases.

The classes in `java.io` are called *blocking* I/O classes, because their methods can block while they are "waiting for the network", in particular the `write` methods of an `OutputStream` and the `read` methods of an `InputStream`, which are used in the `TcpSocketSender` and `TcpSocketReceiver`, respectively. To avoid that calls from the asynchronous Core API block because of the blocking I/O, sending messages is performed asynchronously. The method `sendMessage` serializes the message object to a byte array and stores it in a queue for the endpoint of the destination space URI. The endpoint is the Internet Protocol (IP) address or the host name which can be resolved to an IP address and the port number that identifies a socket address and also an XVSM space. The serialization is performed by a configurable *serializer* which is described below in Section 5.3.3. Sending the serialized messages is performed in tasks, one for each endpoint. This `MessageSendTasks` are executed in a thread pool for the TCP socket transport. They open a connection and send the messages for this endpoint. Failing connect and send attempts are retried and after some time without new messages to send the connection is closed and the task ends.

The `TcpSocketReceiver` has an internal thread with a server socket, that waits for incoming connections on a specified port. For each connection a task to receive messages is created and executed on a thread pool. This is the same thread pool as for the `TcpSocketSender` and it can be configured in the same way as the XP thread pool in the runtime (see Section 4.4.2). The task to receive messages tries to read the serialized message. A configurable serializer that has to be compatible with the serializer on the sender side, deserializes the message and passes it to the `RemoteMessageDistributor`. When a read timeout occurs, the connection is closed and the receive task ends.

As it is described above, the actual transport protocol used by the TCP socket transport is very simple. There is no special handshake or additional connection setup, the messages are simply exchanged when the TCP connection is established. The messages are sent and received concurrently and independent of each other. Each message is sent as an integer number that specifies the length of the following byte array with the serialized message, and then the byte array itself. The exact format of the integer number is defined by the interface `java.io.DataOutput` and is the four bytes of the number in big-endian order (most significant byte first). The format of the serialized message is determined by a serializer as explained next.

5.3.3 Serializer

The *serializer* determines how a message object is represented in the byte array that is sent to another core by a *sender* and how it can be deserialized from such a byte array back to a message object. It thereby defines the wire protocol for the remote communication, when the TCP socket transport is used, together with the length of the byte array as integer number (see previous paragraph). By serializing an object to a byte array stream and deserializing it from there, a serializer can also make a deep copy of objects and can be used in an *entry copier* (see Section 4.4.1).

Five different serializers are currently implemented. One of them, the `JavaBuiltinSerializer` which uses the built-in Java binary serialization, is included in the runtime. The other serializers are in separate modules and can be included when desired (see Section 7.2 for details). Another binary serializer uses the Kryo serialization library [17] because it is small, easy to use and also fast, according to benchmark results in [1]. The Kryo serializer requires classes, whose instances are serialized, to have a no-argument constructor or class-specific serialization code. It can currently not be used for remote communication because not all classes used in messages meet this requirements. However, it can be used to copy entries.

For interoperable serialization, a text protocol or a binary protocol that is supported on various platforms is desirable. Because of the wide support and the existing experience we chose to use XML as basis. MozartSpaces 1.0 has an XML serializer that complies with an XML Schema Definition (XSD) [WF04], which describes the structure of the requests serialized to XML. The serialization to XML itself has been implemented manually, which is cumbersome and difficult to maintain. For the new version the first attempt was made with XStream [25], a library that can serialize arbitrary Java objects to XML and back. However, the XML generated by XStream is schemaless, that is, it does not follow a specific XSD. Furthermore, it is difficult to configure XStream so that the generated XML is valid for a given XSD. Thus we decided to use the Java Architecture for XML Binding (JAXB) [13], start from an XSD and generate the matching Java code.

The structure of this XSD is explained in Section 5.4. We use the JAXB Binding Compiler (xjc) to generate the Java interfaces and classes from the XSD. We do not use the generated Java classes directly in the runtime and CAPI-3, because changes in the XSD would affect the whole MozartSpaces implementation. The xjc code generation is configurable but not flexible enough for our requirements. Especially some requirements regarding thread-safety, for example that generated classes are immutable, cannot be configured. So, instances of the generated Java classes are translated to instances of the Java classes that are used in the runtime and CAPI-3. In contrast to the manual XML serialization of MozartSpaces 1.0, this translation is between Java types, not Java types and strings with XML, and is thus type safe.

The general XVSM XSD that is currently used has “gaps”, because some of the types, for example for entry objects, cannot be defined universally. For this parts of the XML messages we still use XStream, as this serialization is not covered by JAXB and would have to be done manually otherwise. For a concrete use case, when the types of the entry objects are known, these gaps can be filled with the exact XSD statements. Moreover, XStream can also serialize Java objects in the JavaScript Object Notation (JSON) format instead of XML with a simple configuration change. After the built-in binary serialization, the binary serialization with Kryo, the XML serialization with XStream, and the XML serialization with JAXB (and XStream), this is currently the fifth implemented serializer. They are evaluated in Section 8.3. Further serializers can be added by implementing the serializer interface and adapting the MozartSpaces configuration file (see Section 6.3).

5.4 XML Wire Protocol

This section describes the XML wire protocol of XVSM as it is used by the JAXB XML serializer explained in Section 5.3.3. It can be used with the existing TCP socket transport protocol (see Section 5.3.2) or similar transport protocols. It is specified with XML Schema for the request and response messages. These messages consist of the same information as the message objects that are used in the MozartSpaces runtime (see Section 4.1.2). An XML document that is valid for the XVSMP schema contains a message as the only element. Listing 5.1 shows the important parts of the definition for XVSMP and the messages. There we see the XML header with the namespace for the following definitions (`targetNamespace`, short `tns`), the message element, the abstract message and the concrete request message. The response message definition is similar but not shown[1]. The main difference is, that it contains a *response element* instead of a *request element*. For a request message the contained request can be of different types, XSD inheritance is used here like for the messages and in other parts of the schema.

Some of the elements and attributes in the schema are optional. In general, this is the case for every attribute that is not defined with `use="required"` and every element that has a `minOccurs` of `0`, for example, the `answerContainerInfo` element of the `AbstractMessage` in Listing 5.1. For this properties the same default values as mentioned for the messages and requests (Section 4.1.2) and in the core API (Section 6.1) are used.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.xvsm.org/Protocol"
  xmlns:tns="http://www.xvsm.org/Protocol" elementFormDefault="qualified">

  <xsd:element name="message" type="tns:AbstractMessage" />

  <xsd:complexType name="AbstractMessage" abstract="true">
    <xsd:sequence>
      <xsd:element name="answerContainerInfo" type="tns:AnswerContainerInfo"
         minOccurs="0" />
    </xsd:sequence>
    <xsd:attribute name="ref" type="tns:RequestReference" use="required" />
  </xsd:complexType>

  <xsd:complexType name="RequestMessage">
    <xsd:complexContent>
      <xsd:extension base="tns:AbstractMessage">
        <xsd:sequence>
          <xsd:element name="request" type="tns:AbstractRequest" />
        </xsd:sequence>
        <xsd:attribute name="destinationSpace" type="xsd:anyURI" use="required" />
      </xsd:extension>
```

[1]The full schema is included in the published MozartSpaces packages [16].

```
  </xsd:complexContent>
</xsd:complexType>
```

Listing 5.1: XVSMP XSD snippet for request messages

Listing 5.2 shows the request hierarchy for a request to read entries. This hierarchy resembles the hierarchy of the request objects used in the runtime (see Figure 4.3).

```
<xsd:complexType name="AbstractRequest" abstract="true">
  <xsd:sequence>
    <xsd:element name="context" type="tns:PropertyList" minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="TransactionalRequest" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="tns:AbstractRequest">
      <xsd:attribute name="transaction" type="tns:TransactionReference" />
      <xsd:attribute name="isolation" type="tns:IsolationLevel" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="EntriesRequest" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="tns:TransactionalRequest">
      <xsd:attribute name="container" type="tns:ContainerReference"
         use="required" />
      <xsd:attribute name="timeout" type="tns:RequestTimeout" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="SelectingEntriesRequest" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="tns:EntriesRequest">
      <xsd:sequence>
        <xsd:element name="selectors" type="tns:SelectorList" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="ReadEntriesRequest">
  <xsd:complexContent>
    <xsd:extension base="tns:SelectingEntriesRequest" />
  </xsd:complexContent>
</xsd:complexType>
```

Listing 5.2: XVSMP XSD snippet for the request to read entries

Listing 5.3 now shows an example XML message for a read request that is valid for the definitions in the Listings 5.1 and 5.2. It has the request ID `7` on the core `xvsm://localhost:9876` from where it is sent to the core `xvsm://localhost:4242` within an explicit transaction with the ID `2`, that was previously created on that core.

The request tries to read entries from the container with the ID `1` using a selector of type `AnySelector` for a coordinator named `AnyCoordinator` and a timeout of `1000` milliseconds. In the message, the optional element `answerContainerInfo` is not used, so the response is sent back to the virtual answer container of the core `xvsm://localhost:9876`. In the request, default values are used for the request context (`null`), the isolation level (`REPEATABLE_READ`) and the count of the selector (`1`).

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<message xmlns="http://www.xvsm.org/Protocol"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:type="RequestMessage" destinationSpace="xvsm://localhost:4242"
    ref="xvsm://localhost:9876/requests/7">
  <request xsi:type="ReadEntriesRequest" timeout="1000"
      container="xvsm://localhost:4242/containers/1"
      transaction="xvsm://localhost:4242/transactions/2">
    <selectors>
      <selector xsi:type="AnySelector" name="AnyCoordinator"/>
    </selectors>
  </request>
</message>
```

Listing 5.3: Example XML message for a read request

The response message to such a request message contains an *entry response* with a list of entry values and the entry values themselves. Listing 5.4 shows the schema definition for such a response. The *entry values* are the application data objects that a user wrote to the space. Because the type of an application object is not fixed, the only constraint is that it is `Serializable`, it cannot be exactly defined in the schema and thus `xsd:any` is used. This allows to extend the XML document by arbitrary elements. Because of the value `skip` for the `processContents` attribute, they only need to be well-formed and should not be validated when they are processed.

```xml
  <xsd:complexType name="AbstractResponse" abstract="true">
    <xsd:sequence>
      <xsd:element name="error" type="tns:Exception" minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="EntryResponse">
    <xsd:complexContent>
      <xsd:extension base="tns:AbstractResponse">
        <xsd:sequence>
          <xsd:element name="values" type="tns:EntryValueList" />
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="EntryValueList">
```

```
    <xsd:sequence>
      <xsd:element name="value" type="tns:EntryValue" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="EntryValue">
    <xsd:sequence>
      <xsd:any processContents="skip" minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>
```

Listing 5.4: XVSMP XSD snippet for a response with entry values

Listing 5.5 shows an example response message for the the request in Listing 5.3. The entry value that is read is a date object, an instance of `java.util.Date`, and it is serialized with XStream as explained in Section 5.3.3.

```
<message xmlns="http://www.xvsm.org/Protocol"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:type="ResponseMessage" ref="xvsm://localhost:9876/requests/7">
  <response xsi:type="EntryResponse">
    <values>
      <value>
        <date xmlns:ns3="http://www.xvsm.org/Protocol" xmlns="">
          2011-04-07 15:54:59.125 CEST
        </date>
      </value>
    </values>
  </response>
</message>
```

Listing 5.5: Example XML message for a response with entries

5.4.1 Schema Interoperability

The XVSMP XML Schema is currently only supported by MozartSpaces 2.0, there is no further XVSM implementation of the new formal model yet, but we tried to make it as interoperable and extensible as possible. We assessed the interoperability with the XML Schema Patterns for Databinding Detector [24]. Most patterns identified by this tool are marked as *basic* (supported by most XML databinding tools) and only some as *advanced* (less well supported). The detailed detector output shows that the patterns identified in the schema are supported on the .NET platform (xsd.exe version 2.0.50727.42) and by gSOAP, an open source SOAP/XML toolkit for C++. Furthermore, we generated C# code from the schema with the tool xsd.exe version 2.0.50727.1432, included in Microsoft Visual Studio 2008. The generated code looks similar to the Java code generated by JAXB and we expect that it can be used in a similar way.

Additionally, the current schema is based on previous work, especially the schema for the existing XVSM implementations with an interoperable XML serialization between the Java implementation MozartSpaces 1.0 and the .NET implementation XcoSpaces [Kar09]. However, compared to that schema, there were many changes, because of changes in the underlying model or because they make working with the generated code easier. One important change in the new schema is the use of inheritance for the requests, instead of the XSD `choice`, because it is much easier to handle in the conversion between the Java objects. Another one is the usage of XSD `any` for entries and some other elements where the type is not fixed, whereas the previous XVSM implementations support only a fixed set of entry data types for the XML serialization. In the new XVSMP XML we try to support arbitrary application objects as entries.

6 Embedded Core API

After describing the MozartSpaces runtime and how it can be extended, we want to present the core API in this chapter. For a user of the middleware who is not interested in the internals, this API is all that is required to access a space. Section 6.1 describes the basic interfaces of the core API. In Section 6.2 we describe the other API types, the CAPI-3 related classes for coordination, which are not used by the runtime and merely passed to the CAPI-3 implementation, and classes for adding and removing aspects. Before a core is started it can be configured via an XML file or configuration objects. That is explained in Section 6.3. This chapter gives only an overview of the core API that is probably not detailed enough for users of MozartSpaces. Additional documentation, including a tutorial and API documentation generated with Javadoc, is available on the MozartSpaces website [16].

6.1 Basic Interfaces and Requests

The basic interface to access XVSM cores is `MzsCore`. It is shown in Figure 6.1 and its `send` method allows to send a request to the space where it should be processed. This method is overloaded because there are several possibilities to obtain the response.

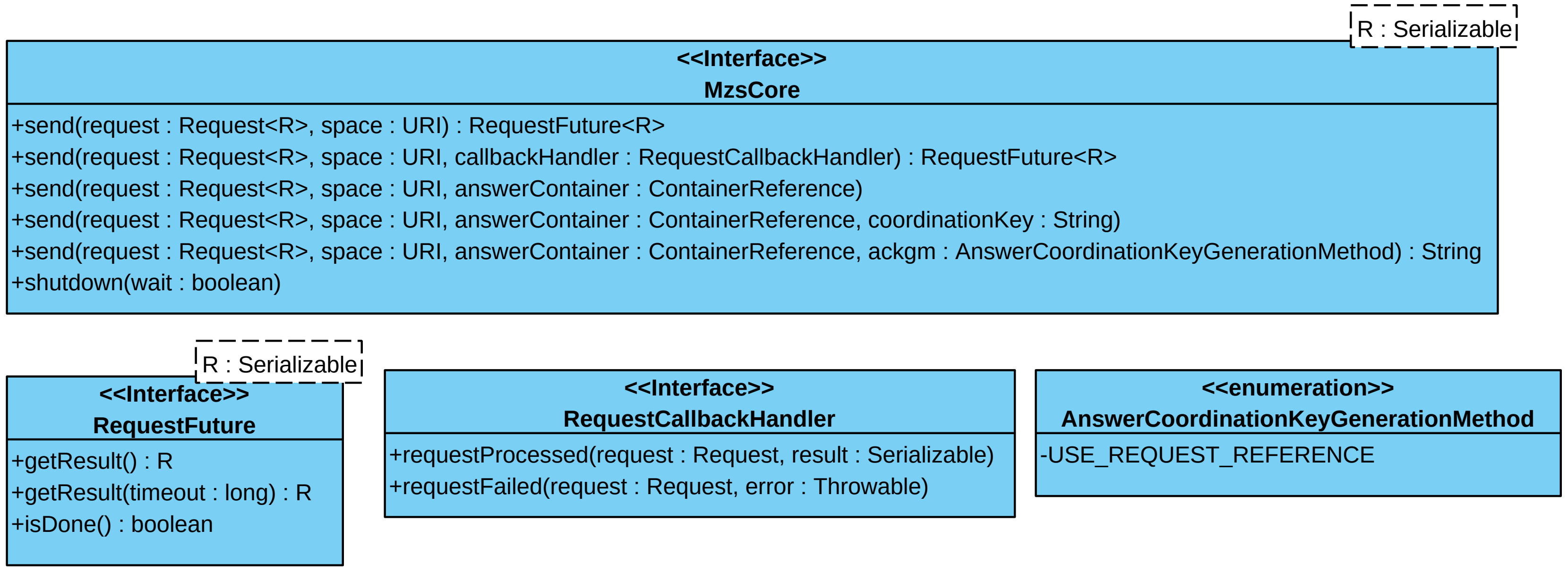

R : Serializable
<<Interface>>
MzsCore
+send(request : Request<R>, space : URI) : RequestFuture<R>
+send(request : Request<R>, space : URI, callbackHandler : RequestCallbackHandler) : RequestFuture<R>
+send(request : Request<R>, space : URI, answerContainer : ContainerReference)
+send(request : Request<R>, space : URI, answerContainer : ContainerReference, coordinationKey : String)
+send(request : Request<R>, space : URI, answerContainer : ContainerReference, ackgm : AnswerCoordinationKeyGenerationMethod) : String
+shutdown(wait : boolean)
R : Serializable
<<Interface>>
RequestFuture
+getResult() : R
+getResult(timeout : long) : R
+isDone() : boolean
<<Interface>>
RequestCallbackHandler
+requestProcessed(request : Request, result : Serializable)
+requestFailed(request : Request, error : Throwable)
<<enumeration>>
AnswerCoordinationKeyGenerationMethod
-USE_REQUEST_REFERENCE

Figure 6.1: Basic core types for the API

6.1.1 Request-Response Calls

With the first two send methods in Figure 6.1, the request is processed and the response is sent back asynchronously. The `RequestFuture` that is returned by this methods represents the result of the request. It also implements the interface `Future` (package `java.util.concurrent`) and provides methods to get the result, optionally with a timeout, or check whether the result is already available. The cancellation of requests is not supported, the method `cancel` of `Future` returns always `false`. The timeout to get the result is only relevant for the future and is not related to any other timeout in the runtime or the request itself. If the request processing failed, an according exception is thrown instead of a result returned. In addition to obtaining the result with the returned future, the second send method allows to specify a `RequestCallbackHandler` with callback methods for the result and a possible error (see Figure 6.1), which are called when a response is available and the result of the future is set.

With the send methods described above the API supports two *invocation asynchrony patterns* [VKZ04]. The `Future`, where the caller decides to get a result or wait for it, implements the *poll object* pattern. The request callback method, where the caller is interrupted when the result arrives, implements the *result callback* pattern. A third pattern, *fire and forget*, can be implemented with one of the send methods described below and a not-existing answer container.

6.1.2 Calls with Answer Container

The three other send methods are for use with an answer container, an XVSM container where the answer (result or error) shall be stored, that is, written with a new request using the FIFO coordinator. Optionally a *coordination key* can be specified, which is used for writing the answer to the answer container with the default key coordinator, where it can later be selected with this key. There is also a send method where the key is “generated” internally, with the currently only option to use the unique request reference as key. So, the answer container needs to have a default FIFO coordinator and optionally, when a key is specified, a default key coordinator.

6.1.3 Creating Requests

The requests that can be sent with `MzsCore` are the same as shown in Figure 4.3. To avoid possible thread-safety issues and changes by aspects, they are not beans with setter methods but mainly *immutable*, that is, properties can only be set when a request is constructed. Therefore, request builders implement a *fluent interface* with *method chaining* [FP10] and shall allow for shorter and easier readable statements. Listing 6.1 shows how some example requests are built, for not specified request properties the default values are set internally.

```
Request<TransactionReference> createTx = CreateTransactionRequest.
  withTimeout(5000).build();

Request<ContainerReference> createContainer = CreateContainerRequest.
  withVerve().name("c1").obligatoryCoords(coords).build();

Request<?> write = WriteEntriesRequest.withContainer(cref).entries(entry).
  timeout(1000).transaction(txref).build();

Request<ArrayList<String>> read = ReadEntriesRequest.<String>withContainer(cref).
  timeout(1000).transaction(txref).build();
```

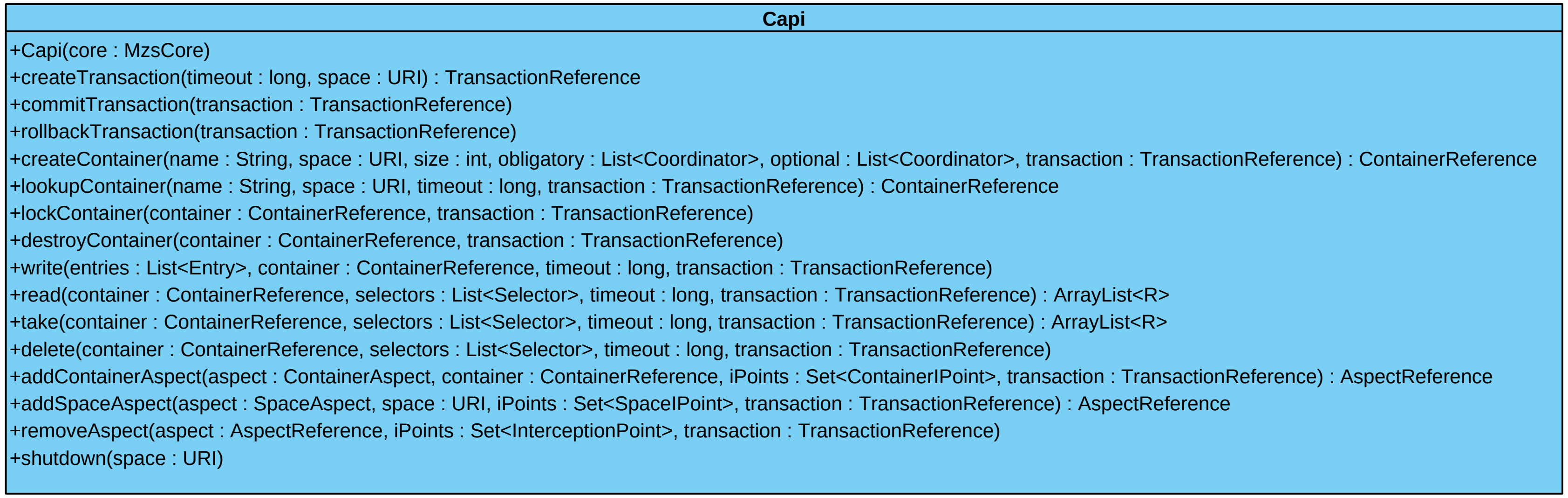
Listing 6.1: Example request builder statements

However, after we implemented some example programs, the request builders seemed to be more verbose and more complicated to use than a non-fluent “big” interface or class (termed *command-query API* in [FP10]) with methods for all implemented requests. Such an interface already exists for the previous XVSM implementations and we also created one MozartSpaces 2.0. It allows synchronous request-response calls. Additionally, we created an asynchronous variant for the other forms of space interaction.

6.1.4 Synchronous Core API Class (`Capi`)

The class `Capi` provides methods for synchronous request-response calls. It is shown in Figure 6.2 and contains a method for each request that can be processed by the runtime[1]. Internally, a request object is created and sent to the space that is either specified explicitly as URI or implicitly with a container, transaction or aspect reference. Then the `getResult` method of the request future is called without a timeout. Therefore, the methods return the result of the processed request or throw an exception if the request processing failed. The methods are overloaded, Figure 6.2 shows the variants with all properties except the request context and isolation level. When request properties are not in the method signature the default values are set internally.

[1]The `PrepareTransactionRequest` is supported by the runtime but there is no method in the `Capi` because it is intended for a future extension of MozartSpaces with distributed transaction support.



```
Capi
+Capi(core : MzsCore)
+createTransaction(timeout : long, space : URI) : TransactionReference
+commitTransaction(transaction : TransactionReference)
+rollbackTransaction(transaction : TransactionReference)
+createContainer(name : String, space : URI, size : int, obligatory : List<Coordinator>, optional : List<Coordinator>, transaction : TransactionReference) : ContainerReference
+lookupContainer(name : String, space : URI, timeout : long, transaction : TransactionReference) : ContainerReference
+lockContainer(container : ContainerReference, transaction : TransactionReference)
+destroyContainer(container : ContainerReference, transaction : TransactionReference)
+write(entries : List<Entry>, container : ContainerReference, timeout : long, transaction : TransactionReference)
+read(container : ContainerReference, selectors : List<Selector>, timeout : long, transaction : TransactionReference) : ArrayList<R>
+take(container : ContainerReference, selectors : List<Selector>, timeout : long, transaction : TransactionReference) : ArrayList<R>
+delete(container : ContainerReference, selectors : List<Selector>, timeout : long, transaction : TransactionReference)
+addContainerAspect(aspect : ContainerAspect, container : ContainerReference, iPoints : Set<ContainerIPoint>, transaction : TransactionReference) : AspectReference
+addSpaceAspect(aspect : SpaceAspect, space : URI, iPoints : Set<SpaceIPoint>, transaction : TransactionReference) : AspectReference
+removeAspect(aspect : AspectReference, iPoints : Set<InterceptionPoint>, transaction : TransactionReference)
+shutdown(space : URI)
```

Figure 6.2: The synchronous core API class

6.1.5 Asynchronous Core API Class (`AsyncCapi`)

The class `AsyncCapi` is the asynchronous variant of `Capi`. The methods it provides are basically the same, but they either return the request future and allow to pass a callback handler or have parameters to specify the answer container and coordination key, dependent on which variant of the overloaded method is used.

6.2 Additional API Classes

The parameters of the interfaces and classes explained above and the properties of requests are also part of the core API. For some of them Java primitive types or standard Java classes like `String` or `URI` are used, the types for others are defined as part of MozartSpaces.

References identify a container (`ContainerReference`), transaction (`TransactionReference`) or aspect (`AspectReference`). They have the same base class with an ID and a space URI. The space URI identifies a core or space instance and the ID has to be unique for a container, transaction or aspect of that space. Internally also requests and corresponding responses use such a reference (`RequestReference`). The reference classes have a string representation of the form `<space-URI>/<reference-type>/<ID>`, for example, `xvsm://localhost:4242/containers/1` for the container with the ID `1` on the space `xvsm://localhost:4242`. This string representation is also used in the XML protocol.

Another MozartSpaces-specific class is the `RequestContext` where arbitrary properties (key-value pairs) can be set. The context of a request is passed to the aspects and coordinators, when the request is processed, and can also be accessed by the runtime. There are different prefixes for request, system, aspect and coordinator properties that should categorize the properties and help to avoid naming conflicts because of equal keys.

Not visible in the class diagrams but of course important are the exceptions used to signal errors. There are different exception classes depending of the type of error. Most of them are checked exceptions and sub-classes of `MzsCoreException`, for example `MzsTimeoutException`, which is thrown when a request or transaction timed out. Unchecked runtime exceptions, instances of `MzsCoreRuntimeException`, are used when an error occurs that usually cannot be dealt with. Such exceptions are used for example for configuration errors of the MozartSpaces core, internal errors that should not occur or when an invalid transaction (already committed or rollbacked, from another core) is used.

6.2.1 Coordination Classes

Coordination classes are the classes for objects used by coordinators in CAPI-3. They are different from the internal coordination classes and specify coordination properties as part of requests. The runtime does not use them and passes them directly to CAPI-3 when an operation is performed. There are classes that represent a coordinator, classes to specify coordination properties when an entry is written and classes to specify properties when entries are selected. A coordinator-specific implementation of the interface `Coordinator` is used when a container is created. A coordinator-specific `CoordinationData` implementation is part of an `Entry` object when entries are written. This object contains the application data object, also called entry value, and the coordination data. An implementation of `Selector`, again coordinator-specific, is used to select the entries that should be read, taken or deleted.

Another coordination-related type that the runtime just passes to CAPI-3 is the isolation level, the possible values are defined by the enumeration `IsolationLevel`.

6.2.2 Aspect Classes

The important types for aspects have already been explained in Section 5.2, namely the aspect interfaces in Figure 5.2 and the interception point enumerations in Figure 5.1. For the aspect interfaces there are abstract and empty implementations in the classes `AbstractContainerAspect` and `AbstractSpaceAspect`. By extending them and overriding the desired methods, aspects can be specified without implementing (empty) aspect methods that are not needed.

6.3 Configuration

Besides the different parameters in the API methods that allow to influence how a request is processed, some components of a MozartSpaces core can also be configured by the user before it is started. The configuration affects mainly the runtime as explained in Section 4.4 and the remote communication explained in Section 5.3. The configuration can be set with a configuration file or programmatically with configuration objects.

When a `MzsCore` instance is created with `DefaultMzsCore.newInstance()`, the configuration is read from an XML configuration file. Listing 6.2 shows a configuration file with the default values for all configurable options. The Apache Commons Configuration library [19] is used for reading the configuration file. It supports the interpolation of variables for configuration options. In the example configuration this is used for the serializer of the TCP socket transport and the space URI. If no configuration file is found, or configuration options are not set in it, internal default values are used.

```
<mozartspacesCoreConfig>
  <embeddedSpace>true</embeddedSpace>
    <coreProcessor>
        <threads>10</threads>
    </coreProcessor>
    <serializers>
        <serializer>javabuiltin</serializer>
    </serializers>
    <remoting>
        <defaultScheme>xvsm</defaultScheme>
        <transports>
            <tcpsocket scheme="xvsm">
                <threads>10</threads>
                <receiverPort>9876</receiverPort>
                <serializer>${serializers.serializer(0)}</serializer>
            </tcpsocket>
        </transports>
    </remoting>
    <spaceURI>${remoting.defaultScheme}://localhost:${remoting.transports.tcpsocket
         .receiverPort}</spaceURI>
    <entryCopier copyContext="false">
        <none />
    </entryCopier>
    <capi3>javanative</capi3>
</mozartspacesCoreConfig>
```

Listing 6.2: Configuration file with default values

A tree of configuration objects is built from the read configuration options and passed to another internal factory method that actually creates the `MzsCore` instance. With the factory method `DefaultMzsCore.newInstance(Configuration config)` a core instance can be created from programmatically created configuration objects.

7 Implementation

The implementation of MozartSpaces is based on the previous chapters. The architecture and design description there is already quite detailed and is implemented at the time of writing, except when otherwise noted. Together with the CAPI-3 implementation described in [Bar10], MozartSpaces is published as open source software under the terms of the GNU Affero General Public License version 3 with additional documentation [16].

MozartSpaces is implemented with the Java Standard Edition 6. We tried to follow the guidelines in [Blo08] for the fundamental use of Java and the Java SE Runtime Environment (JRE) APIs. For the concurrency-related implementation we used mainly [GPB$^+$06] as reference. One notable characteristic result is the use of immutable objects where possible. These are objects whose state cannot be changed after construction and that are implicitly thread-safe. Besides the JRE APIs we used a few open source Java libraries.

7.1 Libraries

Many classes in MozartSpaces are marked with Java Concurrency in Practice (JCiP) annotations [8] to describe whether they are thread-safe. The Simple Logging Facade for Java (SLF4J) [9] is used for logging. It is an abstraction for various logging frameworks. During development and in the examples the Native SLF4J implementation Logback is used. To read the configuration from an XML configuration file the Apache Commons Configuration library [19] is used. For the XML serialization of the XVSMP messages, the Java Architecture for XML Binding (JAXB) [13] implementation included in Java SE 6 and XStream [25] were selected. An alternative binary serializer uses Kryo [17]. Besides these libraries, which are necessary to run applications built with MozartSpaces, further tools and libraries are used for development. They are all integrated into a Maven [Son08] project that is used for managing and building the MozartSpaces packages, documentation and website.

7.2 Maven Project Structure

The MozartSpaces Maven project is divided into several modules. Figure 7.1 shows the important modules of the project and dependencies. If a user wants to create an application with MozartSpaces, she needs at least the runtime module, which includes the core-api and the capi3-api modules. The runtime module also depends on the capi3-javanative module, which is the only CAPI-3 implementation included in the current version. The module capi3-javanative depends on guice, a dependency injection framework used in MozartSpaces as described in [Bar10], and the runtime module depends on commons-configuration (Apache Commons Configuration). Not shown in the figure are modules or dependencies that are required in every module, namely the MozartSpaces module core-common and the library dependencies jcip-annotations, slf4j-api and logback-classic.

The modules runtime, core-api, capi3 (capi3-api and capi3-javanative) and core-common form the MozartSpaces core. Together with their dependencies they are the minimal set of modules that is required to create an XVSM-based application in Java. For notifications the corresponding module needs to be included in application programs, as well as for alternative serializers with JAXB, XStream or Kryo.

Also not shown in Figure 7.1 are modules with only tests or examples, and the wrapper module that allows to use most applications created for MozartSpaces 1.0 with the new version 2.0.

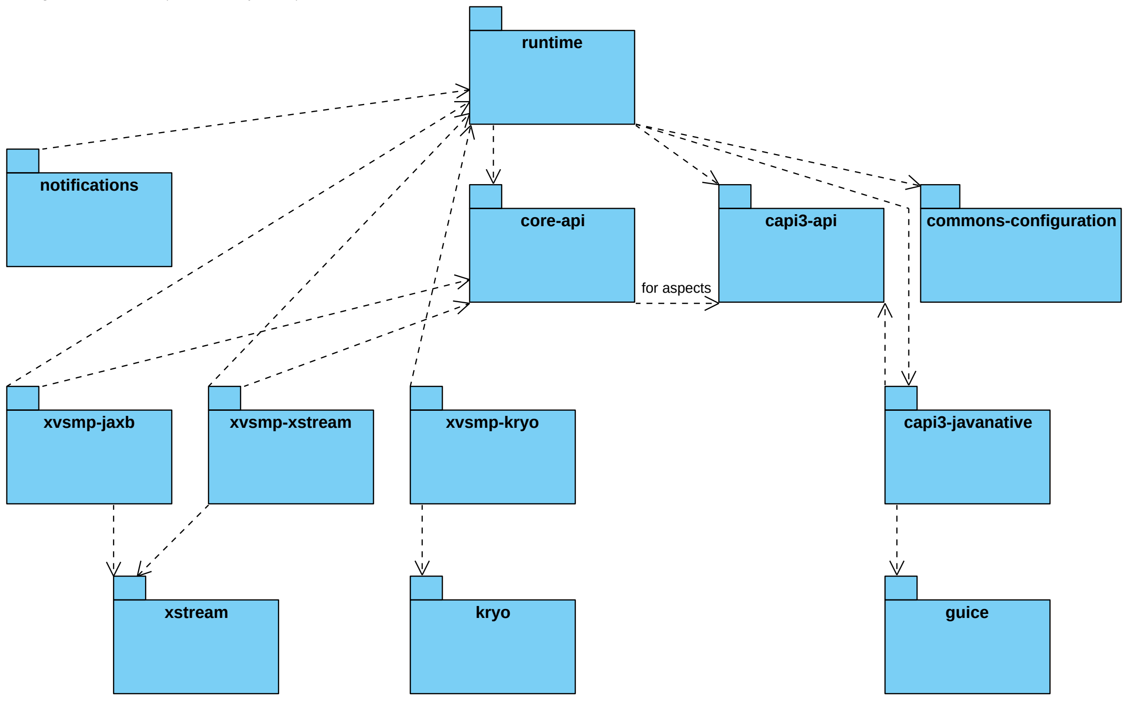


Figure 7.1: Maven project structure—modules and dependencies

The Maven project also includes several plugins that help to improve the software quality or provide statistics. One calculates several source code metrics, including the simple number of “Non Commenting Source Statements” (NCSS). All MozartSpaces modules together, but without the wrapper, examples and tests, have currently 15232 NCSS in 446 Java classes. This also includes the CAPI-3 implementation described in [Bar10]. The biggest modules are capi3-javanative and runtime, followed by core-api.

7.3 Tests and Reviews

Test cases for MozartSpaces are also included in the Maven project. Unit tests are in the same module as the class(es) to test. Integration tests are either in the same module or in extra test modules. JUnit [6] is used for both test types, to mock dependencies we use Mockito [4].

All MozartSpaces modules together, without the wrapper and examples but including CAPI-3, have currently 692 JUnit tests. Besides test cases, the quality of the architecture and source code was examined in reviews.

8 Evaluation

The last chapters described the architecture and implementation of the runtime with remote communication and the API of MozartSpaces. This chapter will now evaluate the performance of MozartSpaces. We measure the time needed for specific operations and configurations. For most tests we use the class `Capi` of the embedded core API (see Section 6.1.4) and include the whole core, the runtime and CAPI-3, in the measurements. We focus on the runtime and do not include detailed coordination benchmarks. Basic coordination benchmarks with CAPI-3 of MozartSpaces 2.0 are described in [Bar10]. The CAPI-3 version used for that benchmarks is an older version than the one used to obtain the results for this chapter. We have made some optimizations in the meantime.

We first describe the evaluation scenario in Section 8.1. Section 8.2 then analyzes the internal processing of the core, how much time the runtime needs compared to CAPI-3 and how much time important phases of the request processing in the runtime need. Section 8.3 shows the performance influence of important configuration options. In Section 8.4 we compare MozartSpaces 1.0 to MozartSpaces 2.0. Section 8.5 briefly shows some scalability results for MozartSpaces 2.0 with a different number of clients and Section 8.6 summarizes the evaluation.

8.1 Evaluation Scenario

The same machine was used for all benchmarks, a laptop with an Intel Core 2 Duo T7500 2.2 GHz CPU, 4 GB RAM, and Windows XP SP3 (32-bit) with all security patches as operating system. All benchmark programs were run with the Java HotSpot Server VM that is part of the Oracle Java SE Development Kit 6 Update 24. The VM arguments used are `-server -Xmx1g`. The first argument selects the server VM, which is relevant because a different just-in-time (JIT) compiler than for the default client VM is used [14]. A JIT compiler compiles portable bytecode into native machine code when a program is executed. The JIT compiler in the Java HotSpot server VM does this in a different way than the one in the client VM. This can affect the execution of programs and thus our measurements. In general, the server VM is intended for long-running server applications which we see as main application area for MozartSpaces. The second VM argument `-Xmx1g` sets the maximum heap size to 1 GB. This relatively high value is used because some benchmark programs write a lot of entries into a space, and cause *out of memory errors* (OOME) otherwise.

We used MozartSpaces 2.0 r10328 for all benchmarks. Unless stated otherwise, we use an embedded space that is configured to use "0" threads for the XP thread pool, that is, no worker threads for request processing as explained in Section 4.4.2. As Section 8.3.1 shows, this is the fastest thread pool configuration. Apart from that, we used the default values, in particular no entry copier, the isolation level `REPEATABLE_READ` and the *javanative* CAPI-3 implementation. An entry copier is not necessary for most use cases and causes some overhead, as Section 8.3.2 shows. The isolation level `REPEATABLE_READ` is defined to be the default in the XVSM formal model. For CAPI-3 the *javanative* implementation is the only option, because the prototype based on a relational database and described in [Bar10] is not included in the current MozartSpaces 2.0 release. The logging level of logback was globally configured to be `INFO`, so that only some information is logged at startup and shutdown but no logging should be performed during the measurements, except when something goes wrong. We accessed the space with the synchronous core API class `Capi` serially in one thread, unless otherwise noted.

We implemented the benchmark programs as JUnit tests with special annotations for JUnitBenchmarks 0.2.1 [15] that we used to measure the execution time. Unless otherwise noted, we used 10 benchmark rounds and 1 warmup round. The warmup round is like a normal benchmark round, but the time needed for it is not included in the measurement. The reason is that the JIT compiler could interfere with the measurement, that is, it could be active while the test case execution is measured, and influence the result. The JIT compiler analyzes the program execution at the beginning and then optimizes (partially recompiles) the native code. This is by default done after 10 000 invocations of a code block. The warmup phase should run the later measured execution paths more often, so that code there is already optimized when the execution is measured and the probability that the JIT compiler recompiles this code again is low. After the warmup, JUnitBenchmarks runs the actual benchmark rounds. Then it calculates the average execution time and the standard deviation (SD). If the standard deviation is higher than 10 % of the average value we mention that and describe whether this can be explained with VM behavior like garbage collector (GC) activity or is caused by specifics of the MozartSpaces runtime implementation.

8.2 Internal Processing

In this section we analyze the internals of the runtime and the overhead compared to CAPI-3. We used one instance of a special test class as entry for all tests. Its code is shown in Appendix A.1. This instance is pre-created, that is, the time for creating it is not included in the measurement. Also pre-created is the Any coordinator and the according selector that we use. This coordinator is implicit and no coordination data is specified when the entry is written. All transactional operations are performed with implicit transactions.

8.2.1 Runtime Overhead

We compare the runtime to CAPI-3 by performing the same space operations from the core API (Capi) and with the CAPI-3 interface (see Section 4.2.6). Our goal is to measure the *overhead* caused by the runtime, that is, the additional time needed for event processing and other components outside of CAPI-3. The CAPI-3 interface has no timeout or aspect support, but we do not use that features in the test cases and the effect in the containers in CAPI-3 is the same. While we have to call only one method of Capi repeatedly and measure the time, at the CAPI-3 layer we have to perform the following steps that the runtime does internally:

1. Create a transaction with the method `newTransaction` of CAPI-3.
2. Create a sub-transaction.
3. Call the actual CAPI-3 operation we want to perform.
4. Commit the sub-transaction.
5. Commit the transaction.

Table 8.1 shows the times for the operations we compare. The values for *create container* show that the runtime does not scale here. We created only 1000 containers, but the time needed by the runtime was already long, much more than for CAPI-3. The SD is quite big (5.60 ±3.04), because each benchmark round takes longer. We found the reason in the event processing, where events are always processed on every container when the (implicit) transaction is committed. Changing that requires refactoring of CAPI-3 and is future work. When we immediately destroy a created container, that is, only one container exists at the same time, which we measure with the *create–destroy* operation pair, we do not see this scalability problem.

For the *create transaction (TX)* operation we got a high SD (1.12 ±0.74) for the runtime value. This is probably because the large number of added transactions, 1 100 000 after the warmup and benchmark rounds, causes the collections in the transaction manager to adapt their size and inserting new entries gets slower. The *create–commit TX* operation pair, where a transaction is created and then immediately committed, does not show this behavior. The time in the table is for creating transactions without a timeout. If we create transactions with a timeout this takes longer (3.08 seconds), because the transactions are added to the timeout processor when they are created and removed from it when they are committed. Rolling back transactions needs equal time as committing them, which can also be seen in Table 8.5, and is therefore not tested here.

For the entry methods we have tested only read and write, there is no difference in the runtime for the other selecting operations (take, delete) for status `OK`. For testing the read operation we created a container, wrote one entry into it and read that entry repeatedly. For testing the write operation we created a container at the beginning of each round and destroyed it at the end. Initially, we did not destroy the container after each round, thus filled it with 1 100 000 entries and got a high SD (1.92 ±0.45 for the Capi and 0.94 ±0.55 for CAPI-3). This is a similar effect as for create TX, but also for CAPI-3 because the collection with all containers is stored there, while the transaction collection is stored in the runtime.

Table 8.1: Comparison of Capi and CAPI-3 interface: average time for 100 000 operations (1000 for create container) in seconds and interface ratio

	API		**Ratio**
Operation(s)	Capi	CAPI-3	Capi/CAPI-3
create container	5.60	0.01	560.0
create–destroy container	2.88	0.85	3.4
create TX	1.12	0.08	14.0
create–commit TX	1.32	0.13	10.2
read entry	1.30	0.24	5.4
write entry	1.44	0.38	3.8

We have also calculated the ratio between the time needed with the runtime and CAPI-3, and included it in Table 8.1. The results show that the runtime overhead factor is between 3.4 and 14, irrespective of create container where a special scalability issue exists. We analyze the runtime performance further in the next section, to find out in more detail where this overhead is spent.

8.2.2 Event Processing and Aspects Overhead

We expected that the main overhead in the runtime is spent for the event processing and the aspects and wanted to analyze that further. This was achieved by commenting out portions of the runtime code for that specific phases and measuring the time for read requests. This is possible because in the tests the requests do not block, there were no aspects used and we know what CAPI-3 returns. We did not use a profiler because we expected it to influence the results. We commented out the calls to the event processing (EP) and the timestamp creation (TS) needed for the event processing separately. We also commented out the calls to the aspect invoker and used a fixed result object instead of actually calling CAPI-3. The detailed description of what we changed in the runtime code is in Appendix A.2. We commented out these calls selectively, not all at once, to see the effect of the individual changes. For example, we commented out the event processing and then measured the time for the read operations. We used the results for the unchanged Capi calls for comparison, obtained with exactly the same benchmark code. Table 8.2 shows the results we obtained. For illustration the results for the CAPI-3 interface alone (see Section 8.2.1) are included in the last line. By measuring the time needed after commenting out a specific portion of the runtime code (column *overall* in the table), we do not directly get the time that would have been needed for it. We estimate this time by subtracting the measured value from the result of the complete read operation (see the column *portion* for the results). We also estimated the duration for creating the timestamps by subtracting the time without event processing ("Capi, EP") from the time without event processing and timestamp creation ("Capi, EP+TS"); this portion cannot be commented out independent of the event processing. The difference to the 100 % in the portions row, 13.3 % of the overall time, is spent in parts of the runtime that are not listed (API, request handler, glue code, ...). The 13.3 % may also be imprecise because we measured the time for the portions indirectly and not in the same benchmark run.

Table 8.2: Estimation of the duration of the phases event processing (EP), timestamp creation (TS), aspect invocation and CAPI-3 call as part of a Capi call: average overall time and phase portion for 500 000 read operations in seconds and percent

	Overall, measured		**Portion, calculated**	
API, excluded phase(s)	in seconds	in %	in seconds	in %
Capi	6.68	100.0	-	-
Capi, EP	6.12	91.6	0.56	8.4
Capi, EP+TS	3.05	45.7	3.63	54.3
Capi, TS	-	-	3.07	46.0
Capi, aspects	5.95	89.1	0.73	10.9
Capi, CAPI-3	5.25	78.6	1.43	21.4
CAPI-3	1.22	18.3	-	-

The results are quite surprising for us. The majority of the time is spent for the event processing and creating the timestamps needed for it. An estimated 46 % of the time is spent only for creating the timestamps. We use the Java API method `nanoTime` in the class `java.lang.System`, the most accurate time source for Java in the standard API according to the documentation, to get the current time. As the impact of having a faster way to get a timestamp would be significant, we want to investigate that as future work.

8.3 Configuration Options

In this section we show the performance influence of important runtime options that can be configured programmatically or with a configuration file (see Section 6.3).

8.3.1 XP Thread Pool

We executed read operations with different values for the number of threads in the XP, that can be configured as described in Section 4.4.2. We used the special value "0", where the thread pool has no worker threads and tasks are executed directly in the calling thread, and the values 1, 2, 5 and 10 for a thread pool with the respective number of worker threads. We prepared the test by creating a container and writing an entry to it that was then repeatedly read. We expected no significant difference between the configurations, because we read serially and only with one thread. The measured values are listed in Table 8.3. These results show that the use of a thread pool with worker threads and the corresponding thread context switch has a significant overhead and increases the time for the test by a factor of around 2.2, for this special case of serial execution where no concurrency is possible. Consequently, the thread pool should be used without worker threads. As Section 4.4.2 explains, this is not possible when the asynchronous API is used and may not be advisable for special transport handlers.

Table 8.3: Comparison of different thread pool configurations: average time for 100 000 operations in seconds

	Number of threads				
Operation	0	1	2	5	10
read	1.28	2.90	2.78	2.81	2.82

8.3.2 Entry Copier

We tested the speed of read and write operations with different entry copiers (see Section 4.4.1). The default configuration uses no entry copier and we compare the result for that configuration to the configurations with an entry copier to see the overhead for the entry copying. The *cloning* entry copier requires the entry class to implement an interface with a clone method, which is implemented in our test entry as listed in Appendix A.1. We tested the *serializing* entry copier with the Java built-in serializer and the serialization library Kryo (see Section 5.3.3). We analyzed read and write operations because there is a difference regarding the entry copying between them: for write the user entry, the application data, is encapsulated in an instance of the class `Entry`, together with the optional coordination data. In contrast, for read only the application data is returned and copied. The results in Table 8.4 show that cloning entries causes almost no overhead; copying entries with a serializer makes the embedded operations several times slower, also for the serializer Kryo that outperforms the built-in serialization. We conclude that immutable objects should be used when possible, because no copying is necessary in that case. If entries should be copied and the speed of the operations is crucial, the cloning entry copier should be used. If speed is not important and implementing the cloning method is too time-consuming or not possible, a serializing entry copier can be used. Kryo is faster than the built-in serialization in our tests, but may require some configuration for custom entry classes (see [17]).

Table 8.4: Comparison of different entry copiers: average time for 100 000 operations in seconds

	Entry copier			
Operation	none	cloning	serializing (built-in)	serializing (kryo)
read	1.27	1.31	5.24	4.43
write	1.46	1.54	5.56	4.69

8.3.3 Transport Serializer

For the remote access we tested the TCP socket transport with different serializers described in Section 5.3.3. For each serializer we created two core instances in the test program. We used one instance as client to access the other core where we repeatedly read an entry and measured the time. The remote communication was performed through the local loopback interface. Figure 8.1 shows the time for the embedded access compared to the Java built-in binary serialization, the XML serialization with JAXB and the XStream serializer with XML and JSON. We also measured the size of the first request and response message, that is, the length of the byte array with the serialized message as it is sent over the socket connection. Subsequent messages may vary slightly in their size for some serializers, because each request has a counter value as ID that is represented as string in XML and JSON messages. The results are depicted in Figure 8.2

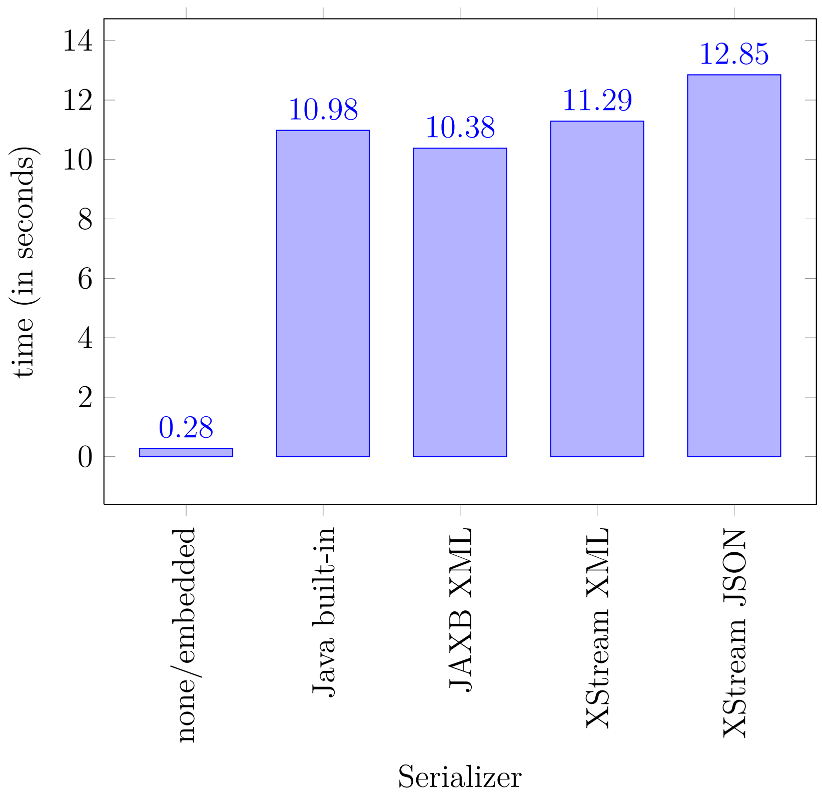


Figure 8.1: Comparison of different transport serializers: average time for 20 000 read operations in seconds

The results show that the XML serialization with JAXB is slightly faster (5.5 %) than the built-in binary serialization. The difference is bigger for the message size, where especially the request message of the built-in serialization is much larger than for JAXB (factor 3.36) or any other serializer. XStream JSON has the smallest message size, but is also the slowest serializer in the test. However, in a scenario with spaces on different machines and a relatively slow network in between, a smaller message size is probably more important than fast (de)serialization. In that case the difference between the embedded and the remote access is probably also much bigger than in our test, where remote access with the fastest serializer (JAXB) took 37 times longer than with embedded access, but without the latency of a real network connection. The performance of the serializer Kryo would also be interesting, because it was faster than the built-in serializer for copying entries (see Section 8.3.2) and the messages are small for the serializer benchmark in [1]. It can

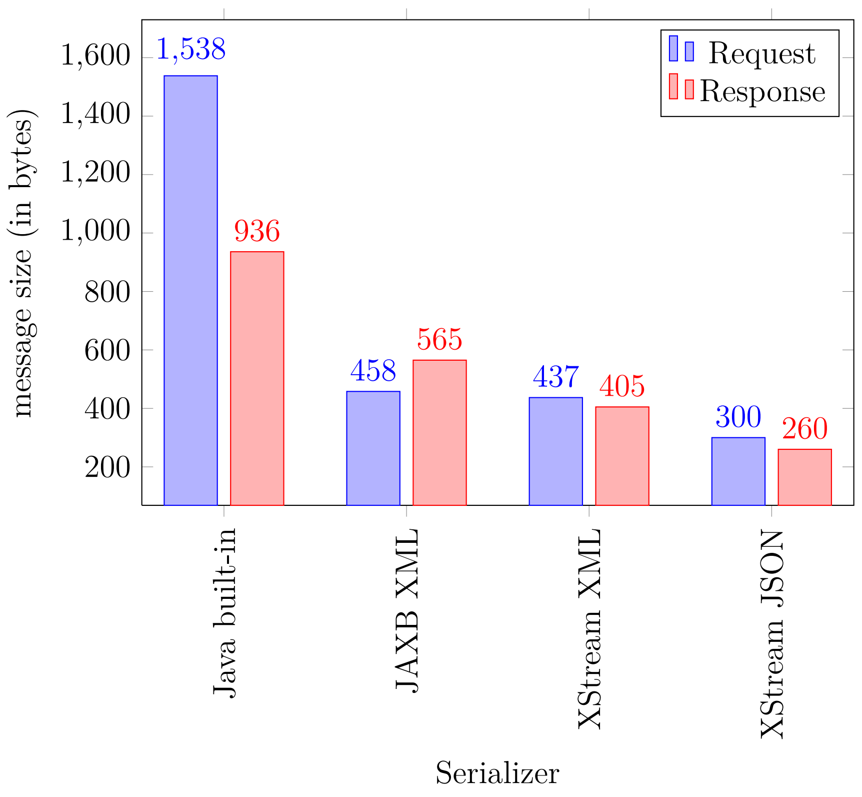


Figure 8.2: Comparison of different transport serializers: size of first request and response message in bytes

currently not be used for remote communication (see Section 5.3.3), but we want to adapt the runtime for it as future work.

Besides the performance also the interoperability can be important for certain use cases. XML and JSON can be easily supported by XVSM implementations on other platforms, while it would be cumbersome to support a binary serialization format like it is used by the Java built-in serialization or Kryo.

8.4 Comparison to MozartSpaces 1.0

In this section we compare MozartSpaces 2.0 to MozartSpaces 1.0 r5212. We used the FIFO coordinator for the tests, because it is predefined for both versions and it is the fastest coordinator in MozartSpaces 1.0 for read operations according to the benchmark results in [Löw08]. The coordinator is implicit and therefore need not be specified when an entry is written. Furthermore, we used the same string of 10 characters for all test entries and accessed the space with implicit transactions. The time for creating the coordinator, selector and entry objects is included in the measured time for all tests.

8.4.1 Embedded Serial Benchmarks

In a first step we performed serial embedded tests for the most important space operations to compare the performance of the two MozartSpaces versions. Table 8.5 shows the results for the container operations. For the *create* container operation we tested 1000 invocations with only 3 benchmark rounds and no warmup, because we soon got an OOME for MozartSpaces 1.0 for more created containers. We have already seen a scalability issue for create container in MozartSpaces 2.0 (see Section 8.2.1), but MozartSpaces 1.0 performs even worse. The results for the *create–destroy* container operation pair shows the same behavior for MozartSpaces 1.0, which indicates a memory leak. The same operation is fast for MozartSpaces 2.0, which is consistent with the results in Section 8.2.1. For the operations that do not scale well we also got high SD values that are included in the table. The container *lookup* operation scales well in both versions, but MozartSpaces 1.0 is faster by a factor of 3.4.

Table 8.5: MozartSpaces versions comparison of container operations: average time and standard deviation for 1000 (create and create–destroy) or 100 000 (lookup) operations in seconds

	MozartSpaces		**Ratio**
Operation(s)	Version 1.0	Version 2.0	1.0/2.0
create	4.43 ±1.66	1.03 ±0.54	4.3
create–destroy	1.94 ±0.76	0.21	9.2
lookup	3.52	1.04	3.4

Table 8.6 shows the results for the transaction operations. We measured an increasing time needed for creating transactions without committing or rolling them back in the meantime. We already experienced that for MozartSpaces 2.0 in Section 8.2.1, but it seems to be worse for MozartSpaces 1.0. We got an OOME with 10 benchmark rounds and measured the time for only 5 rounds then. For the operation pairs *create–commit* and *create–rollback* transaction, where only one transaction is active at the same time, we see constant execution times with MozartSpaces 2.0 being 2.5 times faster than MozartSpaces 1.0. Additionally, we can see that there is practically no performance difference between committing and rolling back a transaction. However, the transactions were not used for any transactional operations in our tests, which is unusual for a real use case.

For the entry operations we tested the *read* and *write* operations and the operation pairs *write–delete* and *write–take*. We tested the read and write operations like in Section 8.2.1 with only one entry for read and a fresh container for each round for write. For the operation pairs we also used one entry, which was removed

Table 8.6: MozartSpaces versions comparison of transactions operations: average time and standard deviation for 100 000 operations in seconds

	MozartSpaces		Ratio
Operation(s)	Version 1.0	Version 2.0	1.0/2.0
create	2.75 ±0.31	1.11 ±0.75	2.5
create–commit	3.25	1.31	2.5
create–rollback	3.25	1.30	2.5

from the container by the second operation immediately after it has been written. The results are listed in Figure 8.3. MozartSpaces 2.0 is constantly faster than MozartSpaces 1.0, by a factor between 6.4 and 6.9.

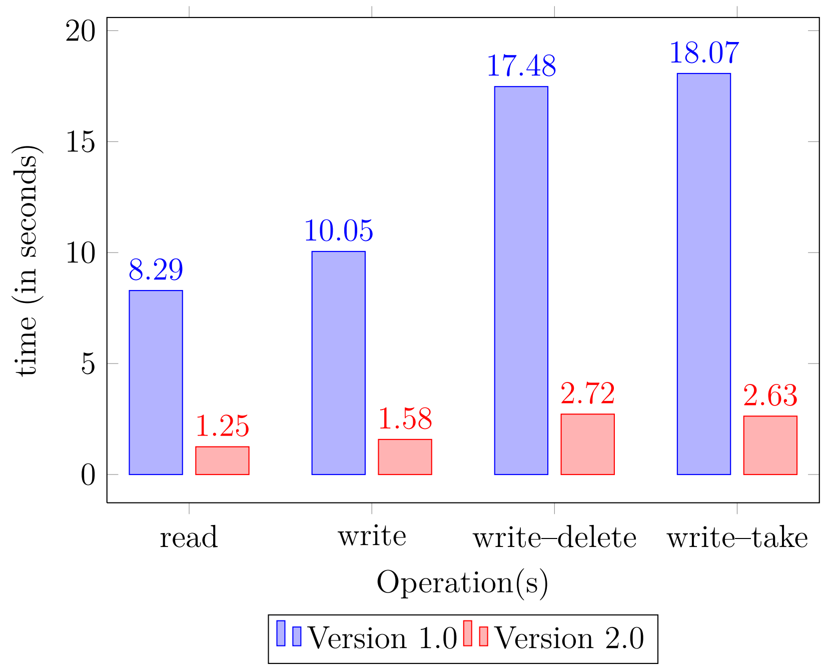


Figure 8.3: MozartSpaces versions comparison of entry operations: average time for 100 000 operations in seconds

8.4.2 Remote Serial Benchmarks

We also wanted to execute the tests described above with a remote space, but that turned out to be rather difficult. The tests for MozartSpaces 1.0 hang without an exception when they were executed. With the JDK tool `jconsole` we saw that the Capi call was stuck in the method `wait` of the class `VirtualAnswerContainer`. We looked for an error in our tests because MozartSpaces 1.0 has been quite excessively used with remote access. We found out, that with the HotSpot client VM instead of the server VM the tests ran more stable, and assume that there is a race condition in MozartSpaces 1.0. We obtained the results in Figure 8.4 with the VM argument `-client` to select the client VM (and `-Xmx1g` like for all other instances and tests) for the MozartSpaces 1.0 server instance and the tests. We started the MozartSpaces 2.0 server instance as planned with `-server`. In contrast to the serializer tests in Section 8.3.3 the remote core ran in its own process. Otherwise the remote access was similar and the local loopback interface with the Java built-in binary serialization was used.

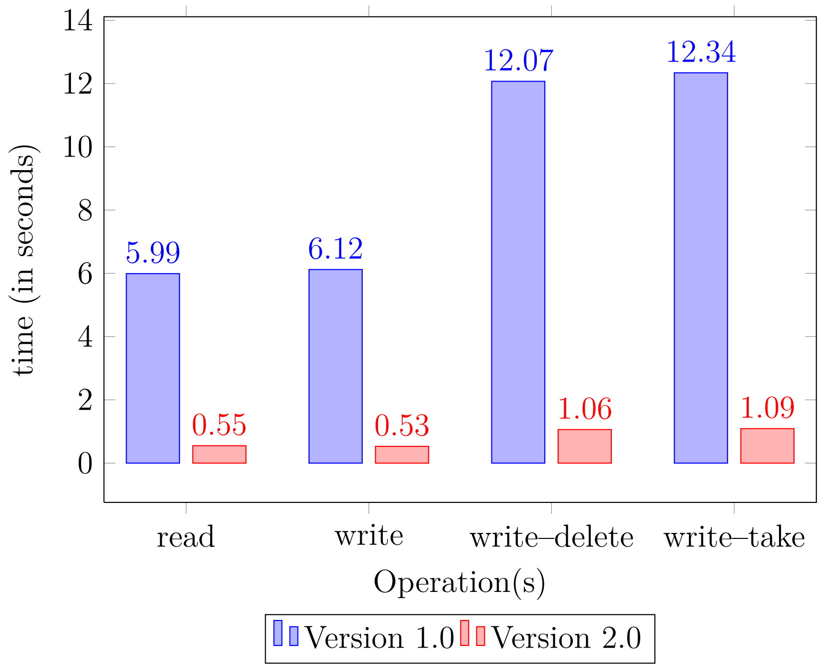


Figure 8.4: MozartSpaces versions comparison of remote entry operations: average time for 1000 operations in seconds

Because of the instability and the low speed of MozartSpaces 1.0 we performed the operations only 1000 times per round. First we had high SDs for the measured values, probably because of the JIT compiler. Therefore we tried different settings for the benchmark and warmup rounds until we found a way to run the tests and get low SDs. The results for *read* and *write* in Figure 8.4 were measured the second time we started the tests for an operation without restarting the server instances, which we restarted after each tested operation otherwise. The values for the *write–delete* and *write–take* operation pairs were measured the first time we ran the tests of an operation for MozartSpaces 1.0, and the second time for MozartSpaces 2.0.

If we now finally look at the actual results, we see that MozartSpaces 2.0 is clearly faster than version 1.0, by a factor between 10.9 and 11.5 depending on the operation.

8.4.3 Concurrent Benchmarks

After the serial benchmarks above we performed concurrent benchmarks with the same operations but executed them in 5 threads concurrently. The results in Table 8.7 show that MozartSpaces 2.0 is again consistently faster than version 1.0, as for the serial benchmarks above. Both versions have a speedup and execute the concurrent operations faster than the serial operations (cf. Figure 8.3). We compare the scalability of the two versions in Section 8.5.

Table 8.7: MozartSpaces versions comparison of concurrent entry operations in embedded mode: average time in seconds for 100 000 operations, 20 000 in 5 concurrent threads each

	MozartSpaces		**Ratio**
Operation(s)	Version 1.0	Version 2.0	1.0/2.0
read	6.91	0.82	8.4
write	8.69	1.01	8.6
write–delete	15.65	1.71	9.2
write–take	15.20	1.78	8.5

The results in Table 8.7 are for an embedded space. We were unable to get results for a remote space and MozartSpaces 1.0. This was probably caused by the same race condition that troubled us for the serial tests (see above).

8.5 Scalability

In this section we take a brief look at the scalability of MozartSpaces. We evaluate the scalability with the *speedup*, defined in [Hil90] as the time needed for a task x on one processor divided by the time needed for the same problem on n processors, or as formula:

$$speedup(n, x) = \frac{time(1, x)}{time(n, x)}$$

We do not have n processors but one processor with n cores and a number of threads. For our test machine these threads share the two cores. So the informative value of our results for more than 2 threads is limited because they do not really show the speedup that would be possible for a CPU with more cores. Because of this hardware limitation, the best possible speedup for our concurrent tests compared to the serial tests is 2, irrespective of the number of threads we use.

Figure 8.5 shows the speedup for the serial entry operation tests in Section 8.4.1 compared to the concurrent tests with 5 threads in Section 8.4.3. We see that MozartSpaces 2.0 scales better than MozartSpaces 1.0 for all operations. The speedup of MozartSpaces 2.0 is between 1.5 and 1.6. Because the tests for the comparison of the MozartSpaces versions are only with 1 and 5 threads, we performed further tests with other numbers of threads that are described next.

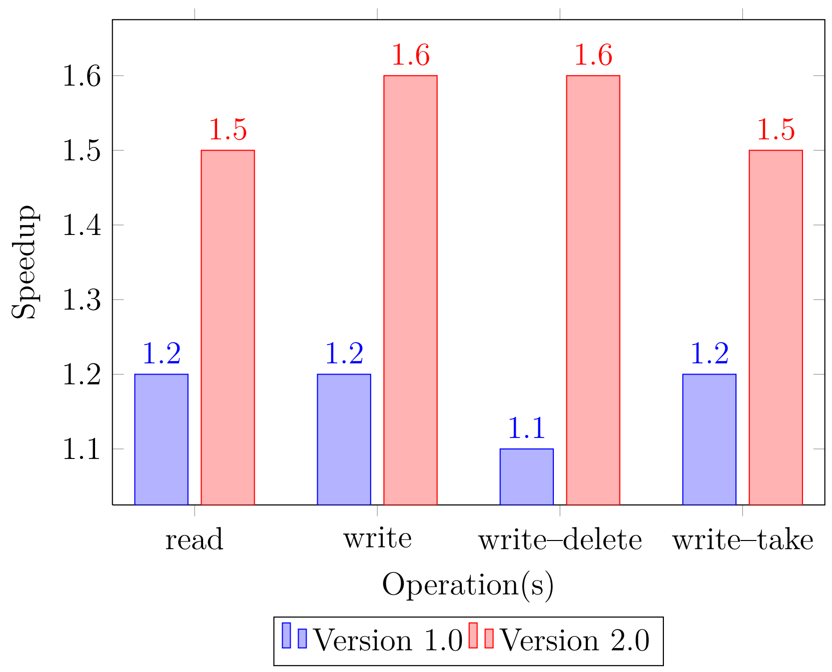


Figure 8.5: MozartSpaces versions speedup comparison of concurrent to serial entry operations in embedded mode (values from Figure 8.3 and Table 8.7)

Table 8.8 shows the speedup for 100 000 concurrent read and write operations with a different number of threads. Each of the n threads performed one n-th of the operations. We get the maximum speedup for 2 threads, 1.8 for the read operation and 1.7 for write. For more threads the speedup is lower, probably because of overhead for the thread scheduling of the JVM and the operation system, which is minimal for 2 threads on the 2 cores of the test machine. The measured times used to calculate the speedup are listed in Appendix A.3.

Table 8.8: Comparison of a different number of application threads: speedup for 100 000 operations with MozartSpaces 2.0

	Number of threads					
Operation	1	2	3	4	5	10
read	1.0	1.8	1.7	1.6	1.6	1.4
write	1.0	1.7	1.6	1.6	1.6	1.5

8.6 Summary

One of the major requirements for MozartSpaces 2.0 was a better performance than for MozartSpaces 1.0. Particularly the runtime should process requests faster and provide better scalability. The tests results which were presented in the last sections show that these requirements are fulfilled. The comparison with MozartSpaces 1.0 shows a superior performance for all measured aspects. The processing of transaction, container and entry requests needs significantly less time. This applies to the embedded mode where the space runs within the application, and also the remote mode where it is accessed over a TCP socket connection. Moreover, version 2.0 of MozartSpaces scales better than version 1.0. It shows a higher speedup for all operations.

We have analyzed the performance behavior for important phases of the request processing in the runtime and tested the influence of several configuration options. We have seen a scalability issue of the event processing in the runtime when the space has a large number of containers. The reason for it has been identified, the required changes for eliminating it are subject to future work. The relatively high portion of the time that is needed to get the timestamps for managing the blocking behavior surprised us and we want to investigate that further and look for faster alternatives.

While we have conducted some tests to assess the performance of the runtime, further benchmarks of MozartSpaces could be performed. We have not measured the performance when requests are blocked and rescheduled. Further evaluation could also include the timeout processing and the asynchronous API. Additional tests could target the predefined coordinators that are implemented in CAPI-3 and assess the memory consumption of the space with different numbers of containers, coordinators and entries. More sophisticated scenarios of concurrent operations could be benchmarked, as well as the notification implementation. However, such an extended evaluation deserves a thesis on its own. This includes also the comparison of MozartSpaces to the XVSM implementation XcoSpaces and other tuple spaces. In [Löw08] the space-based middleware implementations MozartSpaces 1.0 and GigaSpaces were compared to the JBoss Application Server and evaluated with a number of benchmarks. [FWR+05] describes the measurement of throughput and response time for tuple space operations with JavaSpaces as example. This related work could help with the design of a framework for comparing the performance of MozartSpaces 2.0 to other space-based middleware.

9 Future Work

MozartSpaces 2.0 is currently available as alpha version and used for research and teaching. The ITS application mentioned in Section 3.2.1 has been migrated to the new version. It is used in the software stack of the ROADSAFE project [5] where it is part of a disruption tolerant connectivity service [PB11]. MozartSpaces is also the basis for the prototype in the Secure Space project (FIT-IT project 825750). Several profiles will be implemented and extend the space with security functionality. Furthermore, the new version is used by students in a course about space-based computing and for several theses.

The future development of MozartSpaces is driven by the requirements of this research projects and research topics. The next planned extensions are outlined in the following:

- **Performance Improvements:** The performance issues discovered during the evaluation should be fixed. This includes the bad scalability of the event processing for many containers and the large overhead for creating the timestamps. As alternative to the method `nanoTime` to create timestamps, the use of the method `currentTimeMillis` and logical timestamps should be evaluated. Faster remote communication could be achieved with the Kryo binary serializer and the use of the Java non-blocking I/O classes. Additionally, the required refactoring to integrate the second timeout processor will improve the performance for requests with a timeout.

- **Meta Model Implementation:** XVSM meta model support should be added to XVSMP, the runtime and CAPI-3. This would be the basis for runtime monitoring and configuration of the space. A management API, separated from the user API, to get and set meta information should be implemented. An application to monitor and configure SEDA-based applications, including an early version of MozartSpaces, is described in [Laf08].

- **Higher Level APIs:** The low-level interfaces `Capi` and `AsyncCapi` are quite "procedural", there are no objects representing containers nor transactions in the user API. "Proxy" objects around the requests are a more object-oriented alternative to the current low-level interfaces. This could be complemented with more abstract collection views of containers, e.g., a container with a FIFO coordinator could be a `java.util.Queue`.

- **Web Service Interfaces:** Making a space accessible as web service allows for the easy use over HTTP. A RESTful web service interface for MozartSpaces is currently being developed. It can reuse parts of the XVSMP XML schema to represent resources in XML or use a similar representation in JSON. Additionally, a SOAP-based web service could easily be implemented with the use of the XVSMP XML schema for the body part of the SOAP messages.

- **Reliable Communication and additional Transport Services:** The current communication manager should be extended and support reliable communication, that is, resend lost messages, which currently needs to be handled on application level. This requires changes to the TCP socket transport service so that the receipt of messages is acknowledged Additional transport services should be added. This includes the web service interfaces mentioned above and further transport services. [Goi09] describes a *transport manager* with *transport units* for TCP sockets, UDP, Pastry, XMPP and Mule ESB, which were implemented for MozartSpaces 1.0.

- **Lookup, Replication and Persistence Profiles:** These profiles were implemented for the MozartSpaces 1.0 and should be adapted and extended. [Goi09] describes the implementation of a *lookup manager* with *lookup units* for XVSM (lookup container), LDAP, Pastry (DHT) and Gnutella. A very simple replication profile is included in the Pastry lookup. An advanced replication profile is currently being developed for MozartSpaces 2.0. Persistence should be added to the space with its containers, entries and aspects. A persistence profile based on aspects has been developed for MozartSpaces 1.0. For MozartSpaces 2.0 persistence will be integrated directly into the containers of the CAPI-3 implementation.

- **Dynamic Code Downloading:** Classes that are used for entry objects on remote spaces, must be known to the remote JVM. This is usually achieved by having the bytecode of those classes somewhere on the classpath of that JVM. If new entry classes are created and used, their bytecode needs to be transferred to the remote machine, which is cumbersome if done manually. Java allows for the dynamic and remote loading of byte code, a feature which is heavily used by Java RMI. Similarly, this would be possible over our remote communication. However, that approach is limited to the Java platform and is not interoperable. To overcome that shortcoming, a platform-independent representation of classes with their fields and annotations and the ability to dynamically create bytecode on the target platform from that representation is required. Additionally, the bytecode of at least the `hashCode` and `equals` methods should be cross-compiled as well, to get the same matching semantics on all platforms. Such a cross-compiling approach with an XML-based intermediate bytecode format is described in [Pud09].

- **Framework Integration:** Because of its extensibility with coordinators, aspects and profiles, MozartSpaces can be used as integration framework. However, it could be required to integrate it into an existing infrastructure, for example a Java EE application server. The integration of MozartSpaces as Enterprise Information Systems (EIS) with Java EE is possible with the Java EE Connector Architecture (JCA), specifically with *resource adapters* [Sun09]. A MozartSpaces-specific resource adapter would allow access to MozartSpaces from an Application Server (AS), e.g., from within an Enterprise JavaBean (EJB), and also enable the use of components in the AS from space-based applications and XVSM aspects. A basic resource adapter for the integration of the virtual shared memory middleware Corso with an earlier version of the JCA has been developed at the Space Based Computing Group [Mar04]. Integration can also take place with transport services like explained above. Transport services for an Enterprise Service Bus (ESB) like the Mules ESB mentioned above or Apache Camel [21], an open source integration framework, are particularly interesting. A MozartSpaces component for Apache Camel would allow for the sending and receiving of messages with the vast number of middleware systems Apache Camel supports.

10 Conclusion

XVSM is a space-based middleware concept with unique features regarding coordination flexibility and extensibility with aspects. A new formal model with a layered architecture specifies its detailed behavior [Cra10]. The lower layers up to CAPI-3 provide the storage of data entries in containers with transactions and coordinators. They have been implemented as earlier work on MozartSpaces 2.0, the XVSM reference implementation for Java [Bar10]. We described the architecture and implementation of the runtime for MozartSpaces 2.0 that is built upon CAPI-3. It supports blocking operations with timeouts and aspects to extend the middleware functionality. We implemented the event processing with the wait container for blocking operations as described in the formal model and discussed an alternative way to manage events and waiting requests. Additionally, deadlocks of XVSM transactions are detected based on a wait-for graph.

For the remote access a communication manager with a TCP socket transport has been implemented. We developed several serializers that can be used with this transport and specify the concrete wire protocol. One of them is based on an XML schema definition that specifies the interoperable XVSMP. A user API allows embedded or remote access to an XVSM space. It supports synchronous and asynchronous calls with poll objects and callbacks. Additionally, the result of an operation can optionally be stored in a space container (answer container). MozartSpaces supports the publish/subscribe model with notifications that we implemented with XVSM aspects.

The evaluation of the implementation has shown that the runtime fulfills the performance requirements. MozartSpaces 2.0 is considerably faster than the previous version MozartSpaces 1.0 ([Sch08b, Prö08]) for all operations. For example, reading and writing entries in embedded mode (no remote communication) is faster by a factor of 6.6 and 6.4, respectively. For the same operations in remote mode the difference is even bigger with a factor of 10.9 (read) and 11.5 (write). Additionally, MozartSpaces 2.0 scales better than version 1.0. We also analyzed the internals of the request processing in the runtime and measured the influence of various configuration options. We will use the results to improve the runtime. Moreover, extensions are planned for the use of MozartSpaces in research projects.

References

[AvdADtH05] Lachlan Aldred, Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. On the notion of coupling in communication middleware. In *On the Move to Meaningful Internet Systems - 7th International Symposium on Distributed Objects and Applications (DOA)*, 2005.

[Bak03] David E. Bakken. Middleware. In *Encyclopedia of Distributed Computing*. Kluwer Academic Press, 2003.

[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.

[BFK$^+$11] Sandford Bessler, Alexander Fischer, eva Kühn, Richard Mordinyi, and Slobodanka Tomic. Using tuple-spaces to manage the storage and dissemination of spatial-temporal content. *Journal of Computer and System Sciences*, 77(2):322–331, 2011.

[Blo08] Joshua Bloch. *Effective Java*. Addison-Wesley, 2nd edition, 2008.

[CB05] Thomas Connolly and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation and Management*. Addison-Wesley, 4th edition, 2005.

[CES71] Edward G. Coffman, Jr., M. J. Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[CG89] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys (CSUR)*, 21(3):323–357, 1989.

[CKS09] Stefan Craß, eva Kühn, and Gernot Salzer. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *Proceedings of the 2009 International Database Engineering & Applications Symposium (IDEAS)*, pages 301–306, 2009.

[Cou05] George Coulouris. *Distributed systems: concepts and design.* Addison-Wesley, 4th edition, 2005.

[Cra10] Stefan Craß. A formal model of the extensible virtual shared memory (XVSM) and its implementation in haskell – design and specification. Master's thesis, Vienna University of Technology, 2010.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[Emm00] Wolfgang Emmerich. Software engineering and middleware: a roadmap. In *Proceedings of the 22nd International Conference on Software Engineering, Future of Software Engineering Track (ICSE 2000)*, pages 117–129, 2000.

[FHA99] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley, 1999.

[FP10] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages.* Addison-Wesley Professional, 2010.

[FWR+05] Daniel Fiedler, Kristen Walcott, Thomas Richardson, Gregory M. Kapfhammer, Ahmed Amer, and Panos K. Chrysanthis. Towards the measurement of tuple space performance. *ACM SIGMETRICS Performance Evaluation Review*, 33(3):51–62, 2005.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

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

[Goi09] Hannu-Daniel Goiss. Naming and communication management for MozartSpaces – using space based computing on a wider scale. Master's thesis, Vienna University of Technology, 2009.

[GPB+06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice.* Addison-Wesley, 2006.

[GR93] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques.* Morgan Kaufmann Publishers, 1993.

[HBS+02] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. *Java Message Service Specification 1.1.* Sun Microsystems Inc., 2002.

[Hil90] Mark D. Hill. What is scalability? *SIGARCH Computer Architecture News*, 18(4):18–21, 1990.

[Hol72] Richard C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, 1972.

[Kar09] Markus Karolus. Design and implementation of XcoSpaces, the .Net reference implementation of XVSM – coordination, transactions and communication. Master's thesis, Vienna University of Technology, 2009.

[KBM05] eva Kühn, Martin Beinhart, and Martin Murth. Improving data quality of mobile internet applications with an extensible virtual shared memory approach. In *IADIS International Conference on WWW/Internet (ICWI 2005)*, pages 443–450, 2005.

[Kes08] Laszlo Keszthelyi. Design and implementation of the JavaSpaces API standard for XVSM. Master's thesis, Vienna University of Technology, 2008.

[KMG+09] eva Kühn, Richard Mordinyi, Hannu-Daniel Goiss, Sandford Bessler, and Slobodanka Tomic. A P2P network of space containers for efficient management of spatial-temporal data in intelligent transportation scenarios. In *The Eighth International Symposium on Parallel and Distributed Computing (ISPDC 2009)*, pages 218–225, 2009.

[KMK+09] eva Kühn, Richard Mordinyi, Laszlo Keszthelyi, Christian Schreiber, Sandford Bessler, and Slobodanka Tomic. Aspect-oriented space containers for efficient publish/subscribe scenarios in intelligent transportation systems. In *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part I*, pages 432–448, 2009.

[KMM+08] eva Kühn, Richard Mordinyi, Hans Moritsch, Thomas Scheller, and Christian Schreiber. A staged-driven architecture style for a scalable space-based middleware. Technical report, Vienna University of Technology, Institute of Computer Languages, 2008.

[KMS08a] eva Kühn, Richard Mordinyi, and Christian Schreiber. Configurable notifications for event-based systems. Technical report, Vienna University of Technology, 2008.

[KMS08b] eva Kühn, Richard Mordinyi, and Christian Schreiber. An extensible space-based coordination approach for modeling complex patterns in large systems. In *Third International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008)*, pages 634–648, 2008.

[KRL07] eva Kühn, Johannes Riemer, and Lukas Lechner. XVSMP/Bayeux: A protocol for scalable space based computing in the web. In *16th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2007)*, pages 68–73, 2007.

[KRML08] eva Kühn, Johannes Riemer, Richard Mordinyi, and Lukas Lechner. Integration of XVSM spaces with the web to meet the challenging interaction demands in pervasive scenarios. *Ubiquitous Computing and Communication Journal*, Special issue of Coordination in Pervasive Environments, 2008.

[KS05] eva Kühn and Fabian Schmied. Xl-aof: lightweight aspects for space-based computing. In *Proceedings of the 1st workshop on Aspect oriented middleware development (AOMD '05)*, 2005.

[KSC09] eva Kühn and Vesna Sesum-Cavic. A space-based generic pattern for self-initiative load balancing agents. In *Proceedings of the 10th International Workshop on Engineering Societies in the Agents World (ESAW 2009)*, pages 17–32, 2009.

[Küh94] eva Kühn. Fault-tolerance for communicating multidatabase transactions. In *Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS 1994)*, pages 323–332, 1994.

[Laf08] Michael Lafite. Design und Implementierung einer grafischen Komponente für Monitoring von SEDA-Applikationen. Master's thesis, Vienna University of Technology, 2008.

[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(4):457–472, 2001.

[Lec08] Lukas Lechner. A JavaScript API for an eXtensible Virtual Shared Memory (XVSM). Master's thesis, Vienna University of Technology, 2008.

[Löw08] Bernhard Löwenstein. Benchmarking of middleware systems. Master's thesis, Vienna University of Technology, 2008.

[Mar04] Johannes Marchart. Integration von Corso spacebased computing in J2EE. Master's thesis, Vienna University of Technology, 2004.

[Mar10a] 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.

[Mar10b] Daniel Martin. *A Tuplespace-Based Execution Model for Decentralized Workflow Enactment*. PhD thesis, University of Stuttgart, 2010.

[MKS10] Richard Mordinyi, eva Kühn, and Alexander Schatten. Space-based architectures as abstraction layer for distributed business applications. In *Proceedings of the 2010 International Conference on Complex, intelligent and Software intensive Systems (CISIS)*, pages 47–53, 2010.

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

[PB11] Thomas Paulin and Sandford Bessler. A disruption tolerant connectivity service for ITS applications using IEEE 802.11p. In *Proceedings of the 17th European Wireless Conference*, 2011.

[Prö08] Michael Pröstler. Design and implementation of MozartSpaces, the Java reference implemention of XVSM – timeout handling, notifications and aspects. Master's thesis, Vienna University of Technology, 2008.

[Pud09] Arno Puder. Towards an XML-based bytecode level transformation framework. In *Proceedings of the Fourth Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2009)*, pages 97–111, 2009.

[SC11] Vesna Sesum-Cavic. *Self-Organization for Load Balancing and Information Retrieval based on Shared Coordination Spaces.* PhD thesis, Vienna University of Technology, 2011.

[Sch08a] Thomas Scheller. Design and implementation of XcoSpaces, the .Net reference implementation of XVSM – core architecture and aspects. Master's thesis, Vienna University of Technology, 2008.

[Sch08b] Christian Schreiber. Design and implementation of MozartSpaces, the Java reference implementation of XVSM – custom coordinators, transactions and XML protocol. Master's thesis, Vienna University of Technology, 2008.

[Sed03] Robert Sedgewick. *Algorithms in Java, Part 5: Graph Algorithms.* Addison-Wesley Professional, 3rd edition, 2003.

[Son08] Sonatype Company. *Maven: The Definitive Guide.* O'Reilly Media, 2008.

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* Wiley & Sons, 2000.

[Sun09] Sun Microsystems, Inc. *Java EE Connector Architecture Specification, Version 1.6*, 2009.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[TC99] Igor Terekhov and Tracy Camp. Time efficient deadlock resolution algorithms. *Information Processing Letters*, 69(3):149–154, 1999.

[TS07] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2nd edition, 2007.

[VKZ04] Markus Voelter, Michael Kircher, and Uwe Zdun. *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware.* John Wiley & Sons, 2004.

[Wal00] Jim Waldo. *The Jini Specifications.* Addison-Wesley, 2nd edition, 2000.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, 2001.

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

[WF04] Priscilla Walmsley and David C. Fallside. XML schema part 0: Primer second edition. W3C recommendation, W3C, 2004.

[WGH07] Stefan Weinbrenner, Adam Giemza, and H. Ulrich Hoppe. Engineering heterogeneous distributed learning environments using tuple spaces as an architectural platform. In *Proceedings of the 7th IEEE International Conference on Advanced Learning Technologies (ICALT 2007)*, pages 434–436, 2007.

[WMLF98] Peter Wyckoff, Stephen W. McLaughry, Tobin J. Lehman, and Daniel A. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

Web References

[1] Benchmark comparing serialization libraries on the JVM. `https://github.com/eishay/jvm-serializers/`, 2011.

[2] COLLIDE, Faculty of Engineering, University of Duisburg-Essen. SQLSpaces. `http://sqlspaces.collide.info`, 2010.

[3] Dan Creswell. The Blitz Project. `http://www.dancres.org/blitz/`, 2010.

[4] Szczepan Faber and friends. Mockito. `http://mockito.org/`, 2010.

[5] Forschungszentrum Telekommunikation Wien. Robust and distributed safety-improved traffic telematics (ROADSAFE). `https://portal.ftw.at/projects/roadsafe`, 2010.

[6] Erich Gamma and Kent Beck. JUnit. `http://www.junit.org/`, 2011.

[7] GigaSpaces Technologies. GigaSpaces. `http://gigaspaces.com`, 2011.

[8] Brian Goetz and Tim Peierls. Java concurrency in practice (JCiP) annotations. `http://jcip.net/`, 2005.

[9] Ceki Gülcü. Simple logging facade for Java (SLF4J). `http://www.slf4j.org/`, 2010.

[10] IBM. TSpaces. `http://www.almaden.ibm.com/cs/TSpaces/`, 2003.

[11] eva Kühn. eXtensible Virtual Shared Memory (XVSM). `http://www.xvsm.org`, 2010.

[12] Erlend Nossum, Trygve Lie, and Chris McFarlen. SemiSpace. `http://www.semispace.org`, 2011.

[13] Oracle. Java Architecture for XML Binding (JAXB) reference implementation. `http://jaxb.java.net/`, 2011.

[14] Oracle. Java HotSpot Virtual Machine. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html`, 2011.

[15] Stanisław Osiński and Dawid Weiss. JUnitBenchmarks. `http://labs.carrotsearch.com/junit-benchmarks.html`, 2011.

[16] Space Based Computing Group. MozartSpaces. `http://www.mozartspaces.org`, 2011.

[17] Nathan Sweet. Kryo – fast, efficient Java serialization. `http://code.google.com/p/kryo/`, 2011.

[18] Terracotta, Inc. Terracotta. `http://www.terracotta.org/`, 2011.

[19] The Apache Software Foundation. Apache commons configuration. `http://commons.apache.org/configuration/`, 2008.

[20] The Apache Software Foundation. Apache River. `http://river.apache.org/`, 2010.

[21] The Apache Software Foundation. Apache Camel. `http://camel.apache.org/`, 2011.

[22] The Apache Software Foundation. Apache Derby. `http://db.apache.org/derby/`, 2011.

[23] TIBCO. ActiveSpaces. `http://www.tibco.com`, 2010.

[24] W3C XML Schema Patterns for Databinding Working Group. XML schema patterns for databinding detector. `http://www.w3.org/2002/ws/databinding/detector/`, 2008.

[25] Joe Walnes and XStream committers. XStream. `http://xstream.codehaus.org/`, 2008.

[26] Vanessa Williams. Gruple. `http://gruple.codehaus.org/`, 2010.

[27] Zink Digital Ltd. Fly object space. `http://www.flyobjectspace.com`, 2010.

All web references have been last accessed on May 11, 2011.

A Appendix

A.1 Entry Class for Internal Tests

The class shown in Listing A.1 was used for the user entries in the tests in Section 8.2 and Section 8.3. The concrete definition is particularly important for the tests of the entry copiers (Section 8.3.2) and the transport serializers (Section 8.3.3).

```java
public final class TestEntry implements Serializable, MzsCloneable {

    private static final long serialVersionUID = 1L;

    // is set as value for the field "bytes" with ALPHANUM.getBytes();
    public static final String ALPHANUM = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    // is set as value for the field "number"
    public static final int NUMBER = 42;

    private final String key;
    private final Double doubleValue;
    private byte[] bytes;
    private int number;

    // no-arg constructor for Kryo
    @SuppressWarnings("unused")
    private TestEntry() {
        this(null);
    }

    public TestEntry(final String key) {
        this.key = key;
        this.doubleValue = 5.26;
    }

    @Override
    public TestEntry clone() throws CloneNotSupportedException {
         TestEntry clone = new TestEntry(key);
         clone.bytes = bytes;
         clone.number = number;
        return clone;
    }

    // getters and setters omitted
    // standard hashCode and equals methods generated with Eclipse omitted
}
```

Listing A.1: Test class for user entries

A.2 Changes for Measuring Component Overhead

We commented out the following calls or portions of the runtime code to measure the internal overhead in Section 8.2.2:

- **Event Processing**
  - Class `AbstractTask`, method `run`: call of the method `triggerEvents` of the interface `WaitAndEventManager`
  - Class `DefaultTransactionManager`, method `commitTx`: invocation of the method `processTransactionCommit` of the interface `WaitAndEvent-Manager`
- **Timestamp Creation**: invocations of `System.nanoTime`
  - Class `AbstractTask`, method `run`: at the start of the method (set `executionTime` to 0 instead) and before the invocation of the method `triggerEvents` (see above, also the lines in between)
  - Class `DefaultTransactionManager`, method `commitTx`: before the event processing call (see above)
- **Aspect Invocation**:
  - `ReadEntriesTask`: the whole block for the pre- and post-aspect invocations and the condition for skipping the CAPI-3 read because of aspect status `SKIP`; leave the `return` for the entries list
- **CAPI-3**:
  - `ReadEntriesTask`: create a result object (`DefaultReadOperation-Result`) with a list containing an instance of the test entry

A.3 Measurements for Scalability

The results in Table A.1 were used for the calculation of the speedup in Table 8.8.

Table A.1: Comparison of a different number of application threads: average time for 100 000 operations in seconds with MozartSpaces 2.0

	Number of threads					
Operation	1	2	3	4	5	10
read	1.26	0.70	0.75	0.78	0.80	0.92
write	1.51	0.87	0.93	0.92	0.94	1.04