FAKULTÄT FÜR !NFORMATIK

Design and Implementation of the JavaSpaces API Standard for XVSM

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

eingereicht von

Laszlo Keszthelyi
Matrikelnummer 0025953

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuerin: A.o. Univ. Prof. Dr. Dipl.-Ing. eva Kühn
Mitwirkung: Univ.-Ass. Dipl.-Ing. Richard Mordinyi

Wien, 28.10.2008

(Unterschrift Verfasser)

(Unterschrift Betreuerin)

Abstract

Due to the increasing demand to have access to data anywhere at any time, distributed systems are more important these days than they were in the past. Most of the deployed distributed systems have considerable restrictions such as that both communication partners must be aware of each other and running at the same time. These problems are addressed by e.g. space based systems, which allow decoupled communication in time and space via a so called shared space. The JavaSpaces technology developed by Sun Microsystems and the newly developed eXtensible Virtual Shared Memory (XVSM) at the Institute of Computer Languages at the Vienna University of Technology are such space based systems.

This diploma thesis is concerned with the design and implementation of the JavaSpaces API standard for XVSM, using MozartSpaces, the Java based open source implementation of the XVSM model. The implementation shall be realized by developing a "middleman", enabling the collaboration between the JavaSpaces API standard and MozartSpaces. As a result, already existing JavaSpaces based systems and applications may use MozartSpaces without the necessity to adapt or rewrite their source code. Furthermore the implementation shall demonstrate the flexibility and extensibility of MozartSpaces.

Kurzfassung

Aufgrund der steigenden Nachfrage an jedem Ort zu jeder Zeit Zugriff auf Informationen zu haben, sind verteilte Systeme heutzutage wichtiger als sie es noch in der Vergangenheit waren. Viele der eingesetzten verteilten Systeme haben bedeutende Einschränkungen, wie zum Beispiel dass beide Kommunikations-Partner sich kennen und zur gleichen Zeit erreichbar sein müssen. Diese Probleme werden durch z.B. Space-basierte Systeme angesprochen, welche eine Entkoppelung von Zeit und Raum mittels eines so genannten gemeinsamen Space ermöglichen. Die von Sun Microsystems entwickelte JavaSpaces-Technologie und das am Institut für Computersprachen der Technischen Universität Wien entwickelte eXtensible Virtual Shared Memory (XVSM) sind solche auf Space basierende Systeme.

Diese Diplomarbeit befasst sich mit dem Design und der Implementierung des JavaSpaces-API-Standard für XVSM, unter Verwendung von MozartSpaces, der Java-basierenden Open-Source-Implementierung des XVSM Modells. Die Implementierung soll durch die Entwicklung eines “Vermittlers” realisiert werden, der die Zusammenarbeit zwischen dem JavaSpaces-API-Standard und MozartSpaces ermöglicht. Als Ergebnis sollen bereits existierende JavaSpaces basierende Systeme und Applikationen MozartSpaces verwenden können, ohne die Notwendigkeit der Anpassung oder des Neu-Schreibens ihres Quellcodes. Weiters soll mit der Implementierung die Flexibilität und die Erweiterbarkeit von MozartSpaces demonstriert werden.

Danksagung

Ich möchte mich bei Dr. eva Kühn und Richard Mordinyi für die Beaufsichtigung dieser Diplomarbeit bedanken. Ebenfalls möchte ich meinen Dank an Christian Schreiber sowie auch Marian Schedenig aussprechen, die mir immer mit ihren Ratschlägen zur Stelle waren, wenn ich sie gebraucht habe.

Ein besonderer Dank gebührt meiner Familie, die mir während meines gesamten Studiums stets zur Seite standen und viel Geduld mit mir hatten. Nicht zuletzt möchte ich mich auch herzlich bei meiner Freundin Katharina Oremus bedanken für ihre Hilfe, Unterstützung und ihr Verständnis, wenn ich mal eine Nacht durchgearbeitet habe.

Contents

List of Figures

List of Tables

List of Abbreviations

FIFO	First In First Out
IPoint	Interception Point
ITS	Interface Tuple Space
JAXS	JavaSpaces API for XVSM
JS	JavaSpaces
JVM	Java Virtual Machine
LIFO	Last In First Out
LIME	Linda in a Mobile Environment
MARS	Mobile Agent Reactive Spaces
MS	MozartSpaces
P2P	Peer-To-Peer
ReSpecT	Reaction Specification Tuples
RMI	Remote Method Invocation
RPC	Remote Procedure Calls
SBC-Group	Space Based Computing Group
SEDA	Staged Event-Driven Architecture
SOA	Service Oriented Architecture
TCP/IP	Transmission Control Protocol/Internet Protocol
TuCSoN	Tuple Centers Spread over Networks
UDP	User Datagram Protocol
Uuid	Universally Unique Identifier
XQL	XML Query Language
XVSM	eXtensible Virtual Shared Memory

Chapter 1

Introduction

The constant growth of the internet (over 1400 million users in 2008) and the steady distribution of mobile devices makes distributed applications more attractive than ever before. This is due to the various services, which can be used by users independent from their location via their mobile devices, offered by distributed applications. Although, it is a handsome benefit for users, it is a challenge for developers to engineer. They must overcome and hide the drawbacks of mobile devices such as heterogeneous hardware, various operating systems, different connectivity and network failures. In order to simplify the development of distributed applications and relief developers of coping with the mentioned problems, various middle-ware systems have been introduced in the course of time.

The most widespread middle-wares are message passing and remote procedure calls (RPC) systems, but these synchronous communication systems have several drawbacks such as that both communication partners, sending and receiving processes, must know each other, both partners have to be available at the same time and if there is more than receiving processes to be informed, a message must be sent to each of them.

Asynchronous communication systems such as message queues eliminate some of the above mentioned problems by introducing a queue where messages are stored until the receiver(s) are available and willing to accept the message. On the other hand, the sending process still must know in which queue to place the message in order to send it to the right recipients.

Recently middleware systems based on the shared data space paradigm enjoy great popularity because they well address the above mentioned problems. The precursor of such systems is the Linda coordination language [54, 19], which enables the coordination of distributed processes by providing a handful operations that access a shared tuple space. In general, space based middleware offers the decoupling of distributed application participants in time and space.

The JavaSpaces technology [7, 39] is such a popular space based middleware system. It was developed by Sun Microsystems [16] as part of the Jini framework [71, 53, 52, 56], which is based on a service oriented architecture (SOA) [63, 49], in the late 90s. Essentially the JavaSpaces technology is more a service standard API specification with principally predefined interfaces that simplifies the development of space based applications and leaves the actual implementation to the developer.

Another newly developed space based middleware is the eXtensible virtual shared memory (XVSM) [37, 35] developed at the Institute of Computer Languages at the Vienna University of Technology [6]. XVSM provides many features such as various coordination types (First In First Out (FIFO), Last In First Out (LIFO), etc.) and aspects. Should the offered features be insufficient, XVSM offers possibilities to add new or modify existing features too. MozartSpaces(MS) [62, 65, 11], the Java reference implementation of XVSM, provides support for the standard JMS API [61], the JavaScript API [46] and a Java high-level API, but until recently it did not supported the JavaSpaces API standard.

This thesis is mainly concerned with the development of the JavaSpaces API standard for MozartSpaces. It shall prove that it is possible to implement such an API standard with little effort on top of MozartSpaces, using only the resources (the various coordination types, transactions, aspects and the ability to extend MozartSpaces with new features) provided by MozartSpaces. Therefore, the Jini framework and the JavaSpaces API specification have been analyzed to facilitate the best compatible implementation as possible.

The secondary objective of this thesis is the improvement of the existing MozartSpaces implemenetations of the Linda coordination language to enhance its performance. For this purpose the Linda coordination language has been analyzed as part of this thesis, to gain knowledge about possible performance optimizations. A benchmark comparison between the original and the revised implementation of the Linda coordination language should depict the performance difference between both.

The remainder of the diploma thesis is structured as the following: Chapter 2 gives an introduction to the concepts of the Linda coordination model and an overview about existing Linda and Space based implementations. Chapter 3 gives a general view on the Jini framework and its main components, followed by an insight into the JavaSpaces API standard specification in chapter 4. Furthermore chapter 5 deals with an overview on MozartSpaces and various its features. The revised LindaCoordinator as part of this thesis is presented in chapter 6, including the design approach, implementation details and a benchmark comparing the performance of the revised LindaCoordinator to the old one. Chapter 7 is concerned with the main objective of this thesis, the design and implementation of the JavaSpaces API standard as an extension for MozartSpaces. Chapter 8 comprises an evaluation of the implemented

JavaSpaces-MozartSpaces API and a benchmark to compare its performance to the JavaSpaces implementations GigaSpaces, Blitz and Outrigger. Chapter 9 summarizes the results of this thesis.

Chapter 2

Linda Coordination Language

In the year 1985 David Gelernter introduced a new type of concurrent programming, the *generative communication* model [54, 40, 41], better known as the *Linda coordination language*. It is not a particular programming language, it is rather intended as an extension used by any existing programming language.
The Linda coordination language's basis is a "shared data space", which is an abstract computational environment called "tuple space" where *data objects*, called "tuples", are stored or retrieved by diverse processes. This enables among other things that processes don't communicate directly with each other, rather they are using the shared data space to exchange data and coordinate themselves, by placing data objects into or removing them from the space[1]. Therefore the Linda coordination language is not only fully distributed in space but also fully distributed in time [40]. The writing process does not need to wait for a reading process, it simply places its data into the shared data space, then continuing its work. The data objects remains in space until another process, possibly created long after the data object has been placed into the shared data space, withdraws it again. This is also the reason why the Linda coordination language is a generative communication language. The Linda coordination language will be referred to as the "Linda model" in the remainder of this diploma thesis.

The remainder of this chapter is structured as follows: First the basic principles of the Linda model will be described in section 2.1. It is followed by the various terms in section 2.2 and the supplied operations in section 2.3, of the Linda model. Finally section 2.4 gives a survey of several popular implementations of the Linda model.

---

[1] The term "space" is commonly used to refer to the shared data space

2.1 Basic Principles

The Linda model, particularly the data objects and the shared data space, is often described by means of the “Blackboard” model [55], that is most often applied in the field of artificial intelligence, but is applicable in this case too. The Linda’s shared data space is represented by the blackboard and the data objects are represented by notes. Processes can exchange information by putting notes on the blackboard, by reading all the posted notes or even withdraw them to consume or alter the information. The diverse processes, that are collaborating this way, are joining and leaving, which means that the notes on the blackboard exist independently from the processes which created it as long as the notes are not removed. This leads to another important feature of the Linda model: “Anonymous Communication”. The writing process is never aware of which process will read the note it has posted, the same is true for the reading process, which doesn’t know who has posted the note. Certainly it is possible to add additional information to the tuples to specify both the process which has written the tuple and the reading process the note was meant for, but apart from this “extension”, both processes aren’t aware of each other.

2.2 Components

This section describes the various components, including the already mentioned template matching, of the Linda model in detail, in order to provide a foundation for the remainder of this chapter and this thesis.

2.2.1 Tuple

In the Linda model the data objects are represented by data tuples, a fixed length ordered list of typed fields, each possessing an assigned value, for example <“a string”, 3.1415, 42>. A tuple can contain fields of different types. By definition it is not necessary that each field’s value must have a value assigned. This is an important characteristics, because it enables to use tuples with partly unassigned field’s values as a template for retrieving tuples from the tuple space by *template matching*. Templates and the application of template matching when reading from the tuple space will be described later in section 2.2.3 and 2.2.4.
Linda also specifies another type of tuples, the “live tuples” [54]. In contrast to a data tuple, which contains only static data objects, a live tuple can contain both expressions and static data objects, for example <“compute sum”, computeSum(2, 3)> contains the string “compute sum” and the function “computeSum” that sum’s up two integers. After writing the live tuple into

the tuple space, the expression is automatically evaluated. After execution the live tuple turns into an ordinary data tuple that can be queried by processes like any other data tuple.

2.2.2 Tuple Space

In terms of the Linda model the shared data space is called "tuple space" for the simple reason that it is used to either store data or live tuples. The Linda model only specifies the operations that have to be supported by the space, and it is free for the implementation how the tuple space is implemented.

2.2.3 Template

In contrast to conventional programming languages and programming techniques, where elements are accessed by their address, the tuples in the tuple space are accessed by a selection query on its values. Therefore the Linda model specifies that tuples can only be read from the tuple space by using a "template". The template is used by the implementation to find an appropriate matching tuple in the tuple space by comparing the template with all tuples that are stored in the tuple space. Templates are nothing else than tuples but with the exception that they are used only for finding appropriate tuples in the space. The unassigned value of a field is called a "wildcard" when used in a template. Using wildcards in a template permits to find similar tuples in the tuple space.

2.2.4 The Rules of Template Matching

Before the rules of template matching can be explained a few definitions are necessary. A tuple T consists of one or more fields F. The i$^{\text{th}}$ field of a tuple T is denoted by the notation F_T^i. In addition, the i$^{\text{th}}$ fields' type and value are denoted by the notation $F_T^i.type$ and $F_T^i.value$.
The following three criteria [54] must be met, when comparing a tuple T to a template P, in order to decide whether the tuple matches the template or not:

- **size**: Both tuple and template must be of the same size, that is they must have the same number of fields.
- **type**: The corresponding fields F_T^i and F_P^i must be of the same type.

  $$F_T^i.type = F_P^i.type$$
- **value**: If both field's type match, the value of the corresponding fields F_T^i and F_P^i must be matched by one of the following rules:

- If both fields F_T^i and F_P^i have a value assigned then the equation

  $$F_T^i.value = F_P^i.value$$

  must hold else the fields are matching.
- If field F_T^i has a value and field F_P^i has a wildcard assigned, then the fields are matching. In this case the value of F_T^i is assigned to F_P^i.

  $$F_P^i.value := F_T^i.value$$

Table 2.1 depicts some examples of template matching, where the leftmost column contains the templates and the uppermost row examples of tuples that either match the template or not. Matching tuples are denoted by a ✓.

If there are several tuples that match the template, one of them is chosen randomly. On this account the Linda model has a nondeterministic behavior. It might happen that a tuple is never read or taken from the tuple space although there are reading and taking processes which use an adequate template.

Template	<“Apple”, 42>	<“Orange”, 42, 3.1415>
<“Apple”, 42>	✓	—
<“Orange”, 42, 3.1415>	—	✓
<“Orange”, ?int, 3.1415>	—	✓
<?string, 42, 3.1415>	—	✓
<?string, ?int, 3.14>	—	—
<?string, ?int>	✓	—
<?string, ?int, ?float>	—	✓

Table 2.1: Template matching examples

2.3 Tuple Space Operations

The original Linda model offers a simple interface, which specifies 4 operations to operate on the tuple space, two write and two read operations that allow blocking reading. Later the interface has been extended by two additional non-blocking read operations. The operations will be now explained in more detail starting with the write operations and then continuing with the read operations.

Write Operations

out() The *out* operation places an ordinary data tuple T into the tuple space.

eval() The *eval* operation places a live tuple into the tuple space. The tuple space automatically creates a new process for each task performing independently from each other. The result is a data tuple that is stored in the tuple space and can be read. Although eval() is not a write operation in the proper sense, its result is nevertheless stored in the tuple space too.

Read Operations

Both read operations, the *rd* and *in* operation, require a single tuple as parameter that serves as template.

rd() Read a data tuple without removing it from the tuple space. Reading is performed by using a template t to find an appropriately matching tuple. If there is no appropriate tuple available that matches the template the invoking process is suspended until another process creates an appropriate one.

in() The *in(t)* operation behaves in almost the same manner as the *rd(t)* operation except that it removes the tuple from the tuple space.

Because both operations block until there is an appropriate tuple available in the tuple space, two additional operations, *rdP* and *inP*, have been added to the Linda model. These operations are the non-blocking counterpart of the *rd* and *in* operations. If there is no matching tuple available in the space, both operations *rdP* and *inP* return with a failure.

Figure 2.1 shows a typical diagram between 2 different processes. Process A writes a tuple into the tuple space, whereas process B successfully consumes the tuple later.

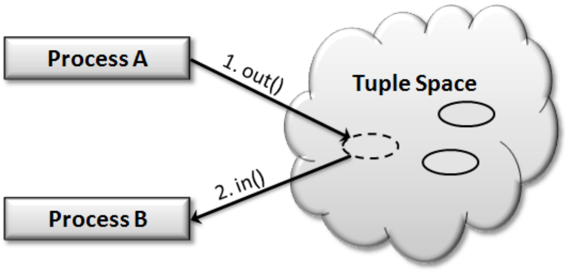


Figure 2.1: A simple tuple space example

2.4 Existing Linda and Space based Implementations

Since the introduction of the Linda model several systems based on the coordination model's principles were implemented in various programming languages. In the following, an overview of relevant implementations of the Linda model with focus on their used concepts and provided extensions is given.

2.4.1 JavaSpaces

The JavaSpaces technology [7] is the Java based implementation of the Linda model by Sun Microsystems.

The concepts of the Linda model have been extended by the advantages of object oriented programming, which allows subtype matching and further by the addition of transactional operations, event notification and leases for registered resources. A detailed description of the JavaSpaces technology is given later in section 4. There are several implementations of the JavaSpaces service available. In the following, the most popular ones will be described.

Outrigger Outrigger is the reference implementation of the JavaSpaces service developed by Sun, which complies with the JavaSpaces specification. The current version implements the JavaSpace05 interface, which also allows the use of the inherited basic JavaSpace interface. Outrigger comes ready to use with the Jini framework. It also supports both a transient and a persistent space services. The latter keeps its state between executions and even preserves the committed entries in case of system failures.

Blitz Blitz [4] is an open source project, developed by Dan Creswell, which implements the JavaSpaces service. It is a fully Jini enabled service that complies to the JavaSpaces specifications implemented on top of Berkeley DB. Blitz aims at making the development and deployment of JavaSpace technologies simpler in order to encourage the spreading of the JavaSpaces technology. It comes as a ready to use package requiring the Jini framework only to run.

Blitz is intended to optimize the storage of entries by fast indexing and hence keep search times to a minimum. The indexing mechanism uses both disk and memory to store entries. This results in a trade-off between performance and reliability. Blitz already provides several predefined configurations each intended to offer balance between data integrity and performance, including a configuration that defines a complete transient behavior and one that defines a complete persistent behavior.

Another mentionable feature of Blitz is that it can be run within the same JVM as the application using it. Running Blitz embedded, circumvents the use of Blitz as a remote Jini service, allowing faster access times. With the possibility to adapt the sources if required and to control the tradeoff between persistence and performance, Blitz invites developers to experiment with it.

GigaSpaces GigaSpaces [5] is a commercial product, developed by a company with the same name, that started as an implementation of the JavaSpaces technology aiming at high scalability and high performance. In the meantime GigaSpaces has provided development frameworks not only for Java but also for .Net and C++. The interoperability between Java, .NET and C++ platforms is realized through XML and is restricted to a specified list of types. The current GigaSpaces version XAP6.6 supports a wide set of APIs, beside JavaSpaces such as JDBC for SQL/IMDB queries. It also contains the OpenSpaces [13] framework, an open source initiative of GigaSpaces, which is build upon the GigaSpace implementation and utilizes the Spring framework [44].
Additional features provided by the current GigaSpaces version XAP6.6 are:

- The integration of external data sources such as Hibernate [31] (by default) or MySQL [30]. External data soruces are deployed for persistancy of GigaSpaces.
- Clustering for scaling, load-balancing and high-availability.
- Security models for:
  - Authentication via name and password.
  - Authorization constrains the execution of specific operations on the space by assigning roles and permissions.
  - Message encryption using SSL.

2.4.2 ActiveSpace

ActiveSpace [1] is a tuple space similar to JavaSpaces, but with a reduced API designed for SEDA [2] style architectures on top of message oriented middleware like Java Message Service (JMS) [48]. The features of such a JMS provider allows ActiveSpace to group spaces and to create subspaces. ActiveSpace is also capable of using the available Quality of Services implemented by a JMS provider, for example:

- Reliable messaging
- Durable (persistent) messaging together with message acknowledgement
- JMS transaction
- XA transactions for the use with other resources like databases

In contrast to the Linda model ActiveSpace does not use templates to query the space, instead a SQL-92 query string is used to filter spaces and the results are placed into a subspace.

---

[2] SEDA (Staged Event-Driven Architecture) [72] is an architecture that decomposes complex, event-driven application into a set of stages connected by queues.

ActiveSpace can operate, similar to JMS, in two different modes: *queue* and *publish/subscribe* based. The former one allows only one consumer to get the object which was placed in the space, while the latter one allows any consumer who is currently subscribed to the space to get the object.
In addition ActiveSpace offers with the implementation of JCache [8] transactional caching and clustered caching.

2.4.3 Blossom

Blossom [69, 70] is a C++ class library, implementing a distributed tuple space that supports the use of multiple tuple spaces. The standard Linda model is extended by additional features [70] in Blossom. The most important ones are the following:

- **Strongly Typed Tuple Spaces:** In contrast to the Linda model, Blossom allows a single tuple space to store only one tuple type. Therefore an application using Blossom might have several tuple spaces, each strongly typed. The advantage is a gain in performance, because the system knows which space contains which specific tuple.
- **Field Access Patterns:** Each field of a tuple additionally contains a flag indicating whether the field is designated as “Actual” or “Any”. The information is used during template matching where *Actual* means that a value must be specified in order to retrieve the tuple field. *Any* allows the use wildcards as specified by the Linda model.
- **Tuple Space Access Patterns:** In Blossom it is assumed that a systems engineer already knows the purpose of the tuple space. Therefore a unique access pattern can be applied to each tuple space on creation, defining the way the tuple space will be accessed and used during runtime. The different access patterns are:
    - **write-once** Tuples can be written only during initialization of the space. Afterwards the tuples are readable only and cannot be removed.
    - **private** The space can be accessed only by one proccess.
    - **write-many** The content of the space is frequently modified.
    - **result** The space’s content is not needed until all data is collected together
    - **synchronization** Two processes can use the space for synchronisation like semaphores. The read operation is not permitted in this access pattern.
    - **migratory** The space can be accessed only by one proccess at the same time.
    - **producer-consumer** Only a single process is allowed to write tuples to the space, whereas several proccess can read or take the entries.
    - **read-mostly** More read than write or take operations are performed on the space.

  - **general read-write** The general access pattern, which does not define any restrictions.

- **Tuple Space Assertions:** Assertions allow to get an insight into tuple spaces, but they are only intended for debugging purposes. This is due to the fact that the whole tuple space must be locked to assure that neither a tuple is added or removed from it during the assertion process.

2.4.4 LIME

LIME (Linda in a Mobile Environment) [60, 51, 50, 10] is a tuple space written in Java that is built on the Linda model but is adapted to the use with mobile agents. Mobile agents are defined by LIME as both physical mobile agents, which reside on mobile hosts that are roaming through the physical space, and logical mobile agents, processes that are migrating from host to host while preserving their code and state.
Each mobile agent has a *interface tuple space* (ITS) bound to itself that is accessible through the conventional Linda operations. The actual content of an ITS is determined on the one hand by the tuples the owning mobile agent is willing to share with others and on the other hand by the content of other ITSs of co-located mobile agents spread across the network. The space created this way forms the “transient shared tuple space”. Each mobile agent has access to the same transient shared tuple space as all the others. The content of the transient shared tuple space is recalculated each time a mobile agent joins or leaves.

The current available LIME version uses LighTS (see section 2.4.5) as its tuple space and offers support for mobile agents based on μCode[3]. Moreover there are two additional versions of LIME available: TinyLIME [28] concentrating on wireless sensor networks and TeenyLIME [27] focusing on for wireless sensor and actuator networks. Since these two implementations are an extension of the original LIME they are not described in detail.

2.4.5 LighTS

LighTS [9, 22, 59] is a lightweight tuple space framework implemented in Java that only provides the basic Linda operations as well as building blocks (e.g. interfaces) to customize and extend its functionalities. This means that the framework is designed as an minimalistic but extensible tuple space that does not support any features like distribution, persistence or security. These features can be implemented using the provided building blocks by applications build around LighTS.

---

[3] μCode[12] provides a minimal set of primitives to support mobility of code and state in Java.

In the first place LighTS was designed as the core underlying tuple space for LIME (see previous section 2.4.4), but its flexible framework makes it possible for other tuple space based systems to use LighTS as their core tuple space adapting it in a way that it meets their requirements.

2.4.6 MARS

MARS (Mobile Agent Reactive Spaces) [25, 26] is a tuple space for the coordination of Java based mobile agents. It extends the conventional Linda model by reactions that can be associated to the space's operations to alter their behavior. Reactions are specified by quadruplets of the form <Rct, T, Op, I>, where Rct is the object containing the reaction that will be triggered by the invocation of operation Op, matching a tuple T, by an agent with identity I. If both the identity I and the tuple T are omitted the reaction will be triggered by each invocation of the associated operation Op. The reaction quadruplets are stored in a "meta-level" tuple space that is associated to the "base-level" space. Each invocation of an operation on the base-level space triggers a meta-level pattern matching mechanism, which finds reactions matching the access event, and executes them.

MARS also provides security on the base of authentication of mobile agents by their identity. Furthermore it is possible to define roles for mobile agents to constrain their access level.

2.4.7 TSpaces

TSpaces [74] is a Java based tuple space system developed at the IBM Almaden Research Center[2]. It is based on the concepts of JavaSpaces but provides a different API to access the tuple space. The API offers several other operations besides the standard JavaSpaces operations write, read and take that facilitate the work with the space like the operation *count* that simply counts the found matching tuples and returns an integer, or *delete* that removes a matching tuple from tuple space but in contrast to a *take* operation does not return it to the process. TSpaces also introduces a new operation called *rhonda* in its API. A process invoking the rhonda operation must pass a tuple as an argument. The tuple's field types are used as a template, which contains wildcards as values only to find a tuple offered by another rhonda operation. If two matching tuples are found they are swapped between both operations and the two involved processes will receive the other one's tuple.

Additionally the latest version of TSpaces already supports the storage of XML documents as tuple fields, which can be queried by using a subset of the XML Query Language (XQL) [18]. TSpaces provides several ways to retrieve a tuple from the space:

- **Match queries:** Match queries further distinguish between:
  - *Structural matching*: Standard template matching as specified by the Linda model, but with subtype matching of each tuple field's type.
  - *Object matching*: Template matching as used by JavaSpaces, where the tuple's type can be a subtype of the template.

  Structural matching is the default matching form used by TSpaces.
- **Indexed queries:** Named fields allow matching of tuples based on a provided field's name and value pair. Additionally, a range of values can be specified to be found by the query in a best effort manner. This means that as many matching tuples as possible should be retrieved.
- **XML queries:** Tuples that contain XML documents in their fields can be queried using a subset of the XQL language.
- **And queries:** Combining two queries with an "AND" operator.
- **Or queries:** Combining two queries with an "OR" operator.

Because both *And* and *Or* queries can be used in a nested way, it is possible to use complex query trees to retrieve tuples.
The functionalities of TSpaces can be extend and modified by defining custom operations that are implemented by so called handlers and factories. Handlers are implementing the operation that is mapped by the factory to an operator.
similar to JavaSpaces, TSpaces provides the specification of leases for stored tuples, the registration for events and the use of transactions to combine various operations into a single atomic operation.
Furthermore TSpaces supports the definition of access permissions that constrain access to the space on operational level based on user identification with name and password. Users are assigned to user groups, which have assigned permissions too.
Depending on its utilization TSpaces can be either used with real database systems (like the IBM's in-house product DB2) or a computer memory based data structure as their tuple storage.

2.4.8 TuCSoN

TuCSoN (Tuple Centers Spread over Networks) [57, 58, 17] is a coordination model intended for the coordination of mobile information agents. Mobile agents are interacting with each other through a local communication space, which consists of multiple *tuple centers*, located at every node of a TuCSoN environment. A tuple center is an extension of the conventional tuple space model, allowing the definition of different behaviors in response to communication events. The behavior of a tuple center is altered through the definition of *reactions* that are associated with the appropriate communication events. Reactions are defined in the specification language ReSpecT (Reaction Specification Tuples) [29, 14].

The execution of an operation triggers all associated reactions at the specific tuple center. In such a case, all reactions are carried out in a single transaction, where either all are performed or none of them. This is realized in such a way that the agent is after all unaware of the various reactions that were triggered by the invocation of an operation.

2.4.9 XMLSpaces

XMLSpaces [67] is an extension of the Linda tuple space model that adds support for storing and querying XML documents in a space, developed at the Technical University of Berlin. XMLSpaces is built on top of TSpaces (see previous section 2.4.7) utilizing its extensibility to facilitate the storing of XML documents as tuple fields by sub-classing the TSpaces' `Field` class. As matching relations XMLSpaces offers several XML query languages such as XQL or XPath, structural matching with respect to a DTD and equality of XML documents.

2.4.10 XMLSpaces.Net

XMLSpaces.Net [68] is a further development of the aforementioned XMLSpaces (see section 2.4.9) and implemented on top of the .Net platform. The basic concepts of XMLSpaces have been adapted and further improved by the .Net implementation. The similarities of XML documents and tuple spaces were taken into account for the implementation. Both share the characteristics of a tree, XML documents by definition and tuple spaces by containing either nested spaces or tuples, whereby the latter might again contain either nested tuples or fields (being the tree's leaves). XMLSpaces.Net considers the tuple space as a single XML document, where tuples are stored as nested XML documents with XML based serial representation of fields containing objects.

XMLSpaces.Net extends the matching capabilities of XMLSpaces by additional features such as *FlatTemplate*- and *DeepTemplate*-matching. FlatTemplate matching considers only the first level without further testing the fields for content equivalence. However, DeepTemplate matching performs a complete recursive matching, considering also the contained sub-tuples.

2.4.11 Corso

Corso [33, 24] is space based middleware developed at the Institute of Computer Languages at the Technical University of Vienna by the space based computing group (SBC-Group) [15]. In contrast to the other tuple space systems, Corso is based on the concepts of *virtual shared memory* [34] that allows the sharing of data objects between processes for coordination or information exchange.

The Corso kernel itself is written in C, but provides several language bindings such as Java, .Net, Ada and C++. Apart from transactions and persistence, Corso supports as well replication of data objects across a network of Corso kernels.

2.4.12 XVSM

XVSM (eXtensible Virtual Shared Memory) [32, 37, 36] is a space based middleware based on the concepts of *virtual shared memory* and developed as well as its predecessor Corso (see section 2.4.11) at the Institute of Computer Languages at the Technical University of Vienna by the space based computing group. It integrates the concepts of P2P-networks [20] to set up a distributed shared space amongst the participating and distributed clients, referred to as *peers*. Tuple spaces are represented in XVSM by containers that have one ore more coordination types, which describe the way how tuples can be stored and accessed by clients, assigned to it. Beside transactions and event notification, XVSM supports aspect-oriented programming [45] to extend and modify the behavior of its space operations on a specific container either before or after the operation is performed.

There are already two open source implementations available, a Java based implementation called MozartSpaces [65] and a .Net based implementation called XcoSpaces [64]. Both implementations are able to interact with each other through a XML-based protocol.

A more detailed description of XVSM and MozartSpaces is given in section 5.

Chapter 3

Jini

The idea of this thesis is to realize the JavaSpaces API on top of the XVSM API. Since JavaSpaces is a part of the Jini framework a detailed description of the framework will be given in the following sections. Jini [21] is a Java based Middleware technology, developed by Sun Microsystems. It is a distributed and service oriented network architecture that enables services and devices to interact without the need of time consuming configuration. Sun refers to this mechanism as “spontaneous networking”, where both hardware and software can be integrated and removed dynamically. Software and hardware components shall “federate” together in a network through the use of Java technology.
In order to achieve this objective, Jini provides an infrastructure of components, which turns the network into a flexible and easy to handle tool that provides a high level of abstraction. The Jini technology is not responsible for managing resources[1], but it facilitates the publishing and discovering of providers of resources, as depicted in figure 3.1. In the terms of the Jini technology the provider of resources is called a “service”.

The Jini technology is a general architecture (see figure 3.2), which provides a simple but powerful API that extends the Java API and is based on Java Remote Method Invocation (RMI), which allows to pass and exchange objects over the network. The Jini API [71] is a collection of service and component specifications, e.g. the JavaSpaces API service and the Jini transaction specification, represented for the most part by interfaces and abstract classes. The Jini architecture distinguishes between *services*, which are forming the central point of the Jini architecture, and *clients*, the users of services, which can be either normal users or even other services. Services and clients are forming together the so called “Jini Community”. Services must register themselves at a lookup service to make it possible for other clients in the Jini Community to find them. A more detailed description of services will be given in section 3.1.

---

[1] Resources can be everything that can be used by clients or even by other services, e.g. a service, computing power, secondary-storage or hardware.

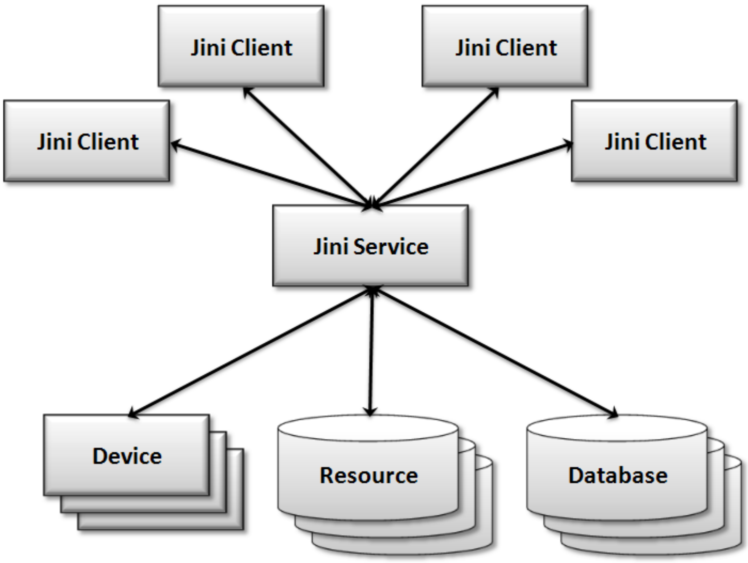


Figure 3.1: Jini service

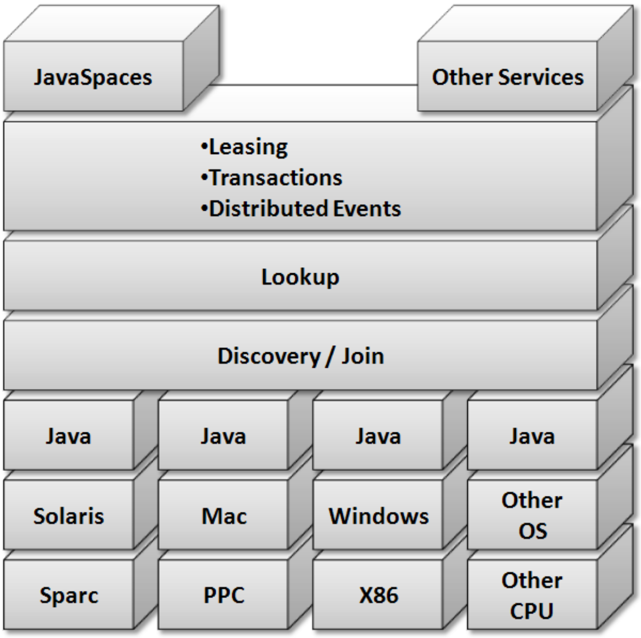


Figure 3.2: Jini architecture

Jini is not just a gathering of diverse services. It allows and encourages writing services that build upon already existing services. In other words a service can be the client of another service and can offer its own service to other services or clients. The term “client” is therefore used in this chapter to refer to clients and services, which are using other services for their operations.
The Jini specification defines also the use of special objects, called *entries*, everywhere, where objects are exchanged and exact-match lookup semantics plays a decisive role. Entries are not addressed directly by naming, but their content is matched against so called templates. More about entries, templates and how template matching is performed will be discussed in section 3.4.

Another essential element of the architecture is the already mentioned *lookup service*. The lookup service is necessary to publish services and accordingly to find a service in the network. It can be considered as a directory of services, where clients can search for a particular service. As the name implies the lookup service is itself a Jini service, with the task to administrate and provide references to other services. Section 3.2 is concerned with the lookup service and its functionality.
Another service that is specified by the Jini specifications is the JavaSpaces service, which is Sun’s implementation of the Linda coordination model on top of the Jini architecture. It is a good example of a service which takes advantage of other Jini services and components. The JavaSpaces’ functionality and its requirements will be described in more detail in chapter 3.3. There are several other services specified by the Jini framework[2] but their description is beyond the scope of this thesis.

The Jini technology predefines further components, which facilitate services and clients to federate and work together, like transactions, notifications and leases. Build upon a “Two-Phase Commit Protocol” [66], the Jini transactions allow clients to perform several operations at different services as a single operation with the guarantee that either all operations will be performed or none of them. The creation of a transaction and the coordination of the two-phase commit protocol between the transaction participants is the transaction manager’s responsibility [53]. The transaction manager itself is a Jini service. More detailed information on transactions will be presented in section 3.6. Distributed events are an important feature of the Jini technology: Clients can register for an event they are interested in and be informed as soon as the event occurred. Details on the functionality of distributed events will be described in section 3.7. ASsuming that due to network failures resources and services cannot be used forever, or due to the mobility of clients unable of assuring their presence permanently, everything in the

---

[2] e.g.: *Lease Renewal Service* - a service that takes care of the renewal of leases on behalf of the lease holder or *Event Mailbox Service* - a service which gathers notifications, allowing the client to obtain them at once when required.

Jini world is leased. Leases can also be considered to be the “garbage collection” mechanism of the distributed Jini system, where resources are removed silently and automatically as soon as the lease expired. Of course, Jini allows the “renewal” of leases in order to show that there is still interest in the resources. Jini leases will be discussed in more detail in section 3.5.

Although Jini is more likely a specification in the form of an API that consists of several interfaces, Sun assists the development of services by providing default implementations of its basic services and components. Towards the end of 2005 Sun decided to rerelease the Jini specification, bundled together with its default implementations of the before mentioned service and component specifications into the “Jini Starter Kit” as version 2.1, which is the latest release by Sun. The new starter kit was not the only improvement Sun made, but also it was released under the Apache License Version 2.0 making the sources available to everyone. As a result, the open source community started a project called “Apache River” [3], with the aim to further develop the Jini framework. The current status of the Apache River project is, that it is undergoing incubation at the Apache Software Foundation to become a part of it.

3.1 Jini Services

Jini services are distributed applications that provide their service to the Jini community by publishing themselves at a lookup service. The lookup service is the only Jini service which does not require registering itself at another lookup service to be findable, however, this is not prohibited.

Services do not send their entire implementation across the network, but rather a proxy that implements a well known interface. The proxy is responsible to enable the client to interact with the Jini service by forwarding the client’s requests to the service. It may also preprocess the client’s data before it is forwarded to the service or post process the service’s data before it is delivered to the client.

Services may implement one or more of the Jini APIs as they require. Such services are called “Jini enabled” and are obligated to comply with the corresponding interface specification, but the implementation itself is not prescribed. This is necessary to ensure that a client, which is looking for a service implementing a specific Jini API, can rely on the fact that it can use the service only on the basis of the well-known API specification.

The Jini framework distinguishes between two types of Jini services: persistent and transient ones. Persistent services maintain their state between executions[3], whereas transient services do not. The former can be run as non-activatable or as activatable service, which means that

---

[3] Their reliability at system failures depends on the implementation of the services.

they are only running when they are needed and not wasting unnecessarily resources while they are waiting. Transient services can only be run as non-activatable services.

The Jini starter kit contains a collection of default implementations of the services specified by the Jini framework. A description of the contained services can be found in appendix A.

3.2 Lookup Discovery Service

The lookup service is one of the key-components of the Jini architecture. The lookup service acts as the central "bootstrapping" mechanism of a Jini system to locate services. It is also comparable to a directory containing an index of several services. A lookup service my contain references to other lookup services, creating a hierarchical structure of lookup services. Another possibility would be to encapsulate other naming and directory services (such as DNS, NIS or LDAP) in lookup services' objects, providing a crossover between Jini lookup services and other naming services.
The bootstrapping mechanism can be separated into two independent procedures:

1. Services making their service available by registering themselves at a lookup service.
2. Obtaining a reference to a service by looking it up at a lookup service.

Both procedures have in common that the client or service has to find a lookup service and obtain a reference to it. Therefore, Jini offers a network-level protocol, used by Jini's runtime infrastructure, that offers two ways to obtain the reference to a lookup service: *multicast*- and *unicast*-discovery. A multicast-discovery is a broadcast, particularly a UDP[4] packet, across the network, where it is expected that all lookup services respond that are available and within the range of the multicast discovery. The client may choose one of the many lookup service remote object references it received. In contrast the unicast-discovery assumes that the hostname and the corresponding port-number of a specific lookup service are known. In this case it is tried to connect directly via a TCP/IP[5] connection to the known lookup service, and on success a remote object reference to the lookup service is received.

The above introduced procedures of the bootstrapping mechanism will be described in detail in the following, to provide a basis for the later implementation of the XVSM JavaSpaces API (see chapter 7).

---

[4] User Datagram Protocol (UDP) [66]

[5] Transmission Control Protocol/Internet Protocol (TCP/IP) [66]

3.2.1 Service Registration

The procedure of a service registration at a lookup service is depicted in figure 3.3 and described below:

1. Find a lookup service through multicast- or unicast-discovery.
2. Receive the remote object reference of the found lookup service.
3. Join the lookup service by handing over the service's proxy and additional attributes wrapped in *entry* objects, which offer more information about the service that might be helpful to a client to find the desired service.
4. Receive a lease as a confirmation about successful registration, as a reference to the registered reference of the object and to be able to request the lease's renewal.

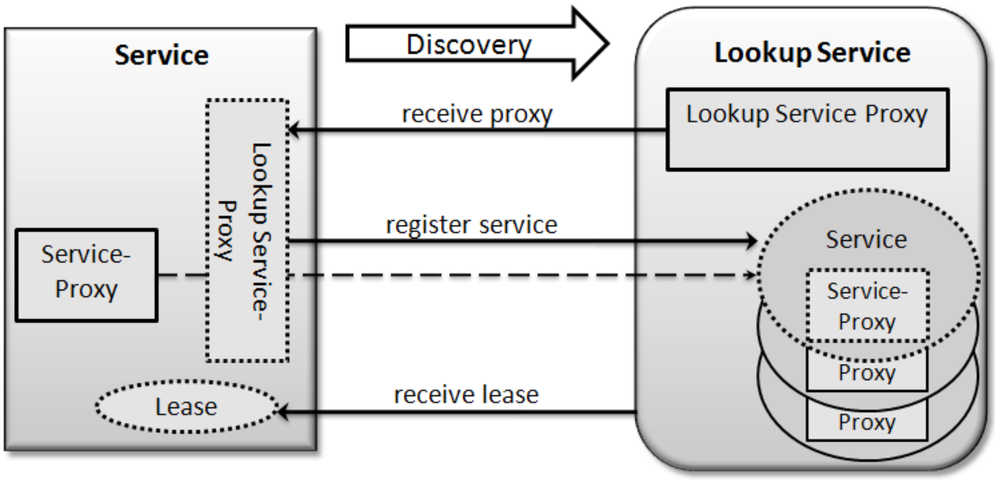


Figure 3.3: Service registration

3.2.2 Service Lookup

The procedure of a service lookup by a client at a lookup service is depicted in figure 3.4 and described below.

Before a client is able to interact with a lookup service it must first obtain a reference to it. It can locate the lookup service via multicast- or unicast-discovery. Like all other services, the lookup service sends back a proxy that implements the `ServiceRegistrar` interface, the so called service registrar object. The client can now "look up" a service by invoking the *lookup* method of the received object. The proxy forwards the request to the lookup service to look up a service that matches in the first place the interface and optionally additional attributes specified by the client. The lookup service will return the proxy of the registered service that

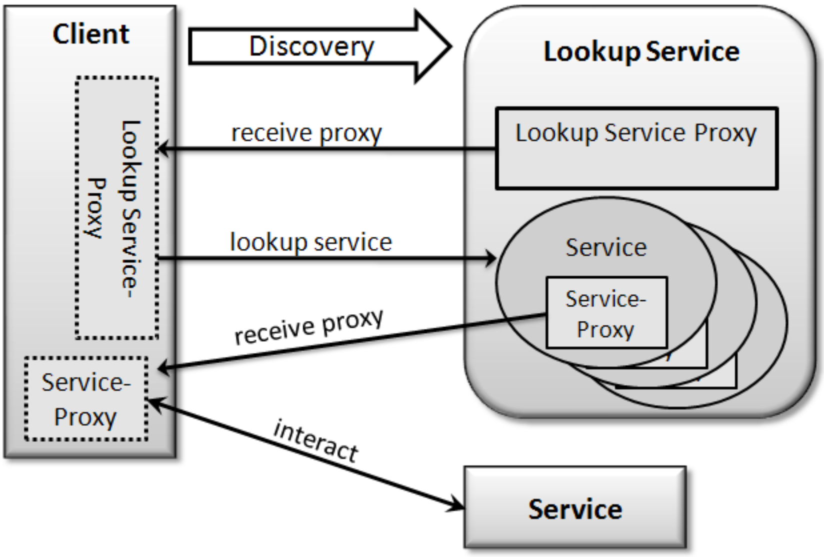


Figure 3.4: Service lookup

matched the specified characteristics. From this moment on, the client interacts with the service via the proxy it obtained by the lookup service.

3.3 JavaSpaces Service

The JavaSpaces service is Sun's implementation of the Linda coordination model on top of the Jini architecture. The JavaSpaces service specification is a good example of a service which takes advantage of other Jini services and components to succeed. The JavaSpaces' functionality and its requirements will be described in more detail in chapter 4.
Jini already provides a default implementation of the JavaSpaces service, called **Outrigger**, in its starter kit.

3.4 Entry

Jini entries are the equivalent of tuples (see section 2.2.1) in the Linda coordination model. The Jini Entry Specification defines that entries are designed to be used in distributed algorithms for which exact-match lookup semantics are useful. An entry is a typed group of objects, which correspond to the tuple's fields, represented by a class that must implement the `Entry` marker interface. In this thesis the term "field" is applied to the fields of an entry which are public, non-static, non-transient and non-final in the context of Jini entries and template matching.

Entries have two applications in the Jini framework:

1. When a service is registered at a lookup server, the registrant can optionally provide sets of attributes which contain additional information about the service. A set of attributes is represented by a class that is a subtype of the `Entry` interface, where the fields represent the attributes. For convenience the Jini Starter Kit already provides classes for the most common attributes (e.g. Name, Version) used for describing services in lookup services.
2. The Jini JavaSpace API specifies to use entries for the objects that are stored in and retrieved from the space. The JavaSpaces services will be described in more detail in see section 4.

The Entry specification prescribes that the fields of Entry objects have to be moved between services and clients in marshalled form[6]. On the client side it is the task of the service's exported proxy to marshall and unmarshall the fields of the entry. The service has to store the entries in the marshalled form, which allows faster exchange between the service and the client, as well as faster matching against templates. If fields of an *Entry* object can not be unmarshalled, an UnusableEntryException will be thrown.

3.4.1 Template and Matching

An entry that is used for matching by retrieval operations is called a template. Template fields can contain either value references or null references as fields of entries, where null-references are interpreted as wildcards by match operations. A template is a match against an entry if the following rules can be applied to the corresponding fields:

- The entry's type must be either the template's type or a subtype of the template's type. In the last case all fields, which are not contained by the template's type, are considered to be wildcards, thus a template can match entries of any of its subtypes. The entry that is returned by a retrieval operation must be of the type or subtype that has matched against the template.
- Matching the value of the corresponding fields of an entry against the fields of a template is similar to the matching of a tuple against a template in the Linda coordination model, see chapter 2.2.3. The fields are compared in their marshalled form and are considered to match if one of the following rules applies:
  - If both fields are non-*null*, they are considered to be equal if the *MarshalledObject*'s equal method returns true.
  - If the template's field is *null*, it is considered to be a wildcard that matches any value, except *null*, of the entry's field.

---

[6] serialized representation of an object in Java

- Although it is not forbidden to store an entry that contains one or more fields with unassigned values, it is not possible to match on `null`-valued fields. This is due to the reason that a `null`-valued field is considered to be a wildcard field.

3.5 Lease

A challenge distributed systems have to meet is how to deal with forgotten, unneeded or out-dated information, which is wasting resources. A garbage collection, in traditional sense, is harder to realize in distributed systems, because to find a way to identify unreferenced or unused resources, is more difficult than in centralised systems. Therefore, the Jini specification introduces another method to deal with out-dated information: it specifies that resources must be leased. A “lease” constrains the time that the holder has to keep the resource, unless it was explicitly removed before the lease expires.
This section introduces the concepts of the Jini Distributed Event specification and the *Landlord* package that is contained in the Jini framework. The landlord package is a semi-finished lease manager, which can be easily used by services to support leases. For this reason and because it was used in the first instance to implement the XVSM JavaSpaces API, the landlord framework will be described more detailed than the Jini Distributed Event specification. A detailed description of the specification is beyond the scope of this thesis and can be found in the Sun’s original Jini Distributed Event specification [71].

3.5.1 Concepts of Jini leases

The Jini specification distinguishes between the “lease holder”, the entity which requests to lease a resource for a specific amount of time, and the “lease grantor”, the entity that decides whether to grant the lease and specifies the maximum lease time. The lease grantor’s decision depends on the defined lease policy, whether a fixed period is always granted or the lease holder request is involved in the decision. In the latter case there might be an upper limit on the lease duration, defined by the lease policy, which the lease holder must not exceed. As soon as the lease expires, the lease grantor will remove the leased resource without sending a “warning” to the lease holder. However the lease holder is able to *renew* the lease at any time, informing the lease grantor that it is still interested in the leased resource. This requires the lease holder to take care of the lease itself. To simplify matters, the Jini framework provides a *Lease Renewal Manager* that is able to renew leases on behalf of the lease holder automatically and silently. The Jini framework specifies a “Lease Renewal Service”, which is a service that offers the functionality of the lease renewal manager, and also provides a default implementation called

“Norm”.

As soon as the lease holder does not require the leased resource any more it can cancel the resource’s lease, causing the lease grantor to take care of removing the leased resource.

If the lease grantor has granted the lease of a resource for a specific period, it returns an object that implements the `Lease` interface. As it is shown in the class diagram in figure 3.5, the `Lease` interface provides amongst other things the methods *renew*, *cancel*, which allow the lease holder to either renew or cancel the lease without contacting the lease grantor directly. The lease object also contains the lease time that was granted by the lease grantor. The lease holder must check the received lease, because the granted lease time can be less or equal to the requested lease duration.

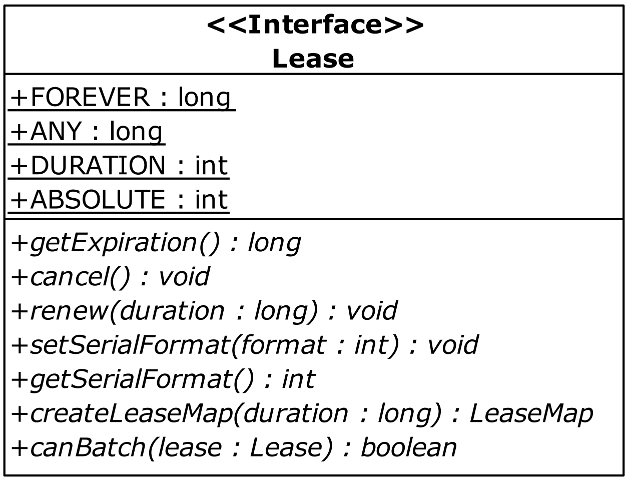

<<Interface>>
Lease
+FOREVER : long
+ANY : long
+DURATION : int
+ABSOLUTE : int
+getExpiration() : long
+cancel() : void
+renew(duration : long) : void
+setSerialFormat(format : int) : void
+getSerialFormat() : int
+createLeaseMap(duration : long) : LeaseMap
+canBatch(lease : Lease) : boolean

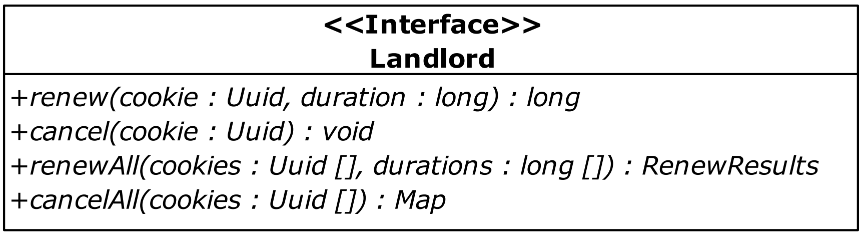
Figure 3.5: Jini Lease interface

3.5.2 Landlord Package

The Landlord package is intended to simplify the provision of leases in Jini services or other Java RMI based distributed systems. To accomplish that the landlord package comes with an almost ready to use solution that consists of a set of interfaces and classes, providing a basis for the later lease’s implementation. A brief description of the most important ones is given below:

Landlord Interface The basic interface that services, lease grantors so called “landlords”, must implement, to be able to utilize the *landlord* package, is the `Landlord` interface, see figure 3.6 for its class diagram. The interface specifies that leases must be identified by a “cookie”, an Universally Unique Identifier (Uuid), internally by the landlord and associated to the leased resources. Furthermore it specifies those remote methods that are needed by the lease object, which the lease holder obtains after registering its resource for a specified amount of time, to communicate back to the lease grantor.

- *cancel & cancelAll*: Cancels the lease that is specified by the provided cookie. *CancelAll* is a batch operation to cancel more than one lease, by providing a list of cookies, with a single call. On failure it returns all leases that failed to be canceled.
- *renew & renewAll*: Renews the lease, specified by the provided cookie, by the requested duration and returns the new lease object. *RenewAll* is a batch operation to renew more than one lease, by providing a list of cookies and the related durations, with a single call.



<<Interface>>
Landlord
+renew(cookie : Uuid, duration : long) : long
+cancel(cookie : Uuid) : void
+renewAll(cookies : Uuid [], durations : long []) : RenewResults
+cancelAll(cookies : Uuid []) : Map

Figure 3.6: Landlord interface

LeasedResource Interface To associate a lease with its resource the landlord package provides the `LeasedResource` interface, see figure 3.7 for the class diagram. It is the correspondent part to the lease object that is handed over to the lease holder, which will remain on the Landlord's side. Hence, it contains the cookie and the expiration of the lease. It is recommended but not required to include at least a reference to the leased resource in the implementation of this interface.

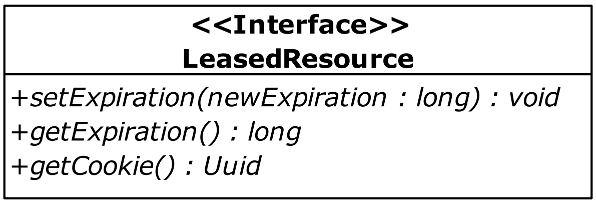


Figure 3.7: LeasedResource class

LandlordLease Class After the lease holder's lease request was granted it receives a receipt, represented by an instance of the `LandlordLease` class, as a confirmation. As depicted in figure 3.8 the `LandlordLease` class extends the `AbstractLease` class and inherits from the `Lease` interface. There is no necessity to extend the `LandlordLease` class, since it is ready for use as is. Amongst other things it allows to verify the lease's expiration and renew or cancel the lease by invoking the remote methods of the landlord interface internally.

LandlordLeaseMap Class A ready to use class that allows to renew or cancel several LandlordLeases together in a batch process. Just as the `LandlordLease` class this class'

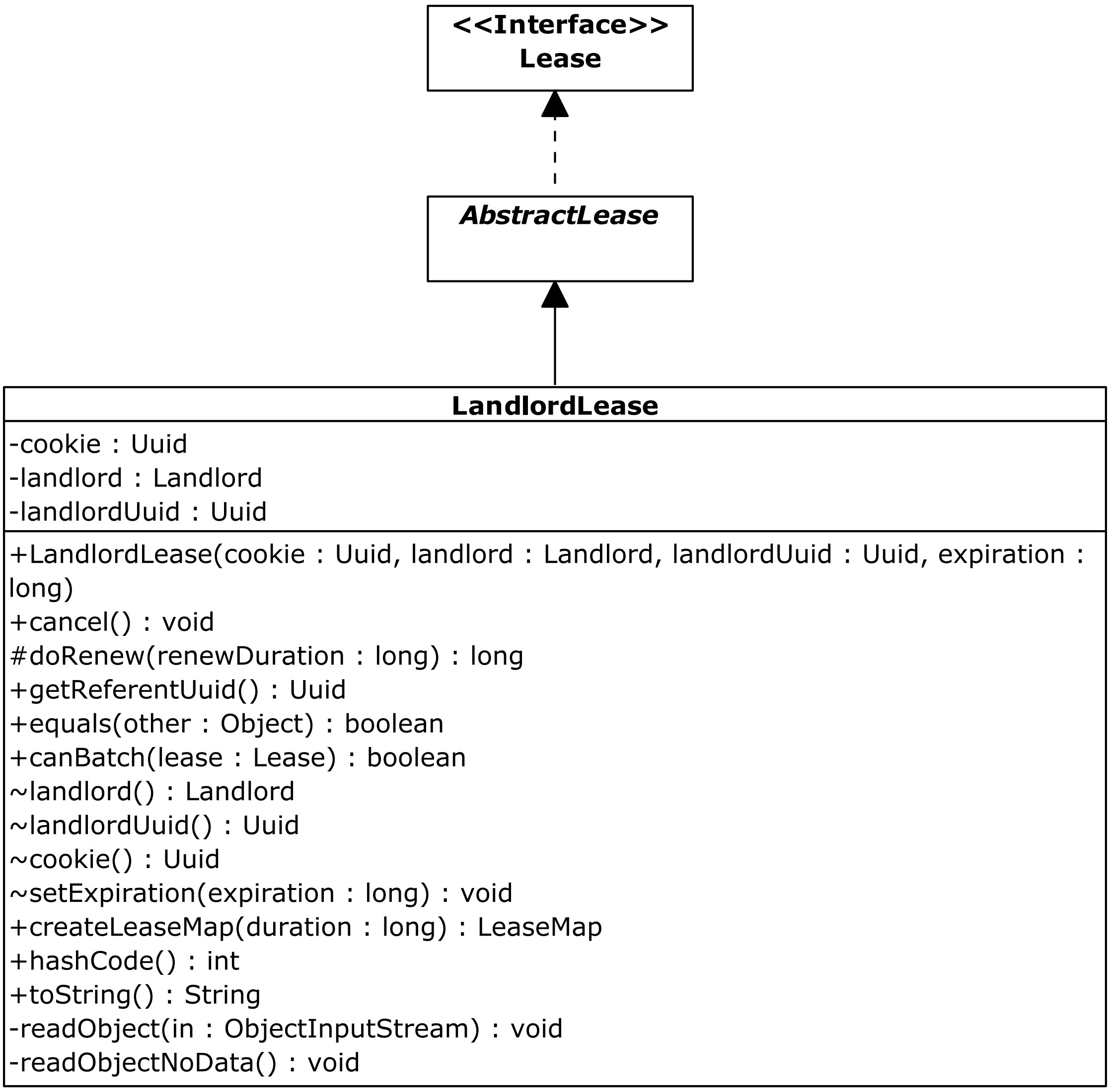

<<Interface>>
Lease
AbstractLease
LandlordLease
-cookie : Uuid
-landlord : Landlord
-landlordUuid : Uuid
+LandlordLease(cookie : Uuid, landlord : Landlord, landlordUuid : Uuid, expiration : long)
+cancel() : void
#doRenew(renewDuration : long) : long
+getReferentUuid() : Uuid
+equals(other : Object) : boolean
+canBatch(lease : Lease) : boolean
~landlord() : Landlord
~landlordUuid() : Uuid
~cookie() : Uuid
~setExpiration(expiration : long) : void
+createLeaseMap(duration : long) : LeaseMap
+hashCode() : int
+toString() : String
-readObject(in : ObjectInputStream) : void
-readObjectNoData() : void

Figure 3.8: LandlordLease class

renew and cancel methods invoke the remote methods of the landlord interface internally.

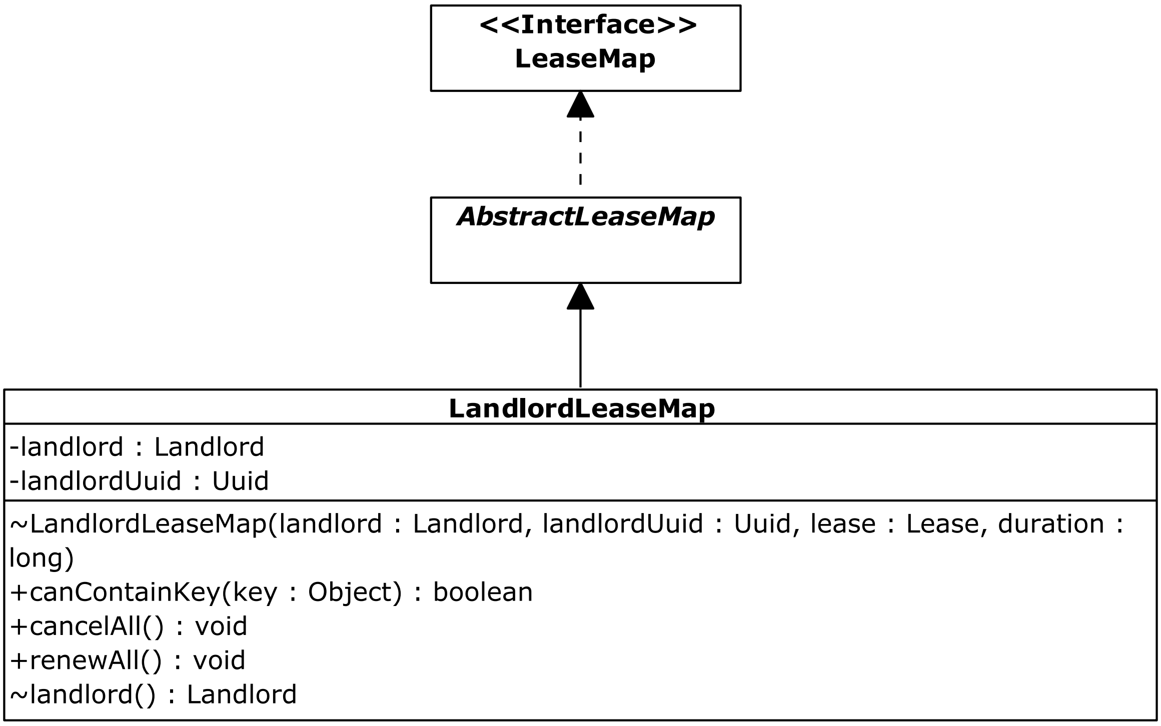


Figure 3.9: LandlordLeaseMap class

LeasePeriodPolicy Interface The `LeasePeriodPolicy` interface is implemented by objects that allow calculating lease grants and renewals on the basis of their implemented lease policy. The interface's predefined methods are *grant* and *renew*, as shown in figure 3.10, both requiring to hand over an instance of `LeasedResource` and the new lease duration.

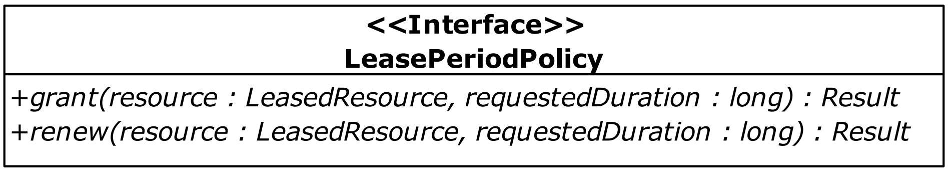

<<Interface>>
LeasePeriodPolicy
+grant(resource : LeasedResource, requestedDuration : long) : Result
+renew(resource : LeasedResource, requestedDuration : long) : Result

Figure 3.10: LeasePeriodPolicy interface

LeaseFactory Class After the lease time was successfully granted by the object that implements the `LeasePeriodPolicy` interface, the `LeaseFactory` class creates a `LandlordLease` class instance that can be returned to the lease holder. Figure 3.11 shows the `LandlordLease` class' class diagram.

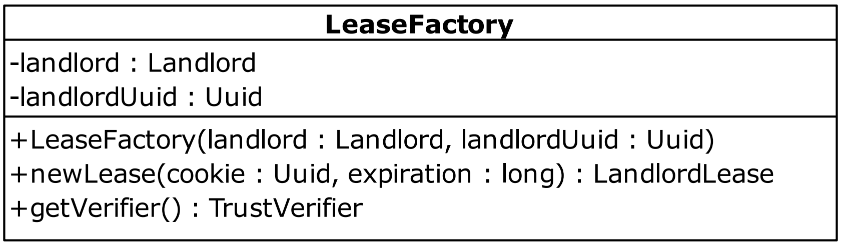

LeaseFactory
-landlord : Landlord
-landlordUuid : Uuid
+LeaseFactory(landlord : Landlord, landlordUuid : Uuid)
+newLease(cookie : Uuid, expiration : long) : LandlordLease
+getVerifier() : TrustVerifier

Figure 3.11: LeaseFactory class

3.6 Transactions

Transactions are essential for distributed systems. They allow to treat a group of operations as a single atomic operation and guarantee that either all operations will successfully complete or none of them. I.e., operations wrapped in a transaction will appear to everyone as if they were executed “simultaneously”.

Since clients can make use of several services at the same time, a client possibly wants operations that belong together, perhaps distributed over various services, to be performed as if they were a single operation. In this case the client relies on the fact that either all operations will be processed or none of them.

Jini transactions have been designed especially to coordinate transactions across multiple services allowing clients to perform related operations on various services as a single atomic operation. Jini does not confine the use of its transactions to Jini services only, any other service can use Jini transactions premising that it behaves in conformity with the specification. Due to the fact that Jini transactions are supposed to support distributed transactions over several services it is build upon a “Two-Phase Commit Protocol”. The protocol requires that services, which are involved in a transaction, vote on a commit request. Based on the result the services will either *commit* (roll forward) or *abort* (roll back) the changes that where made under the transaction. The Jini transactions specification distinguishes three kinds of actors that take part in a transaction:

- The *transaction manager* (see section 3.6.2) is responsible to create, observe and coordinate the progress of transactions.
- The *transaction participant* (see section 3.6.3) , in most instances a service but it can be any process that supports the participant[7] contract by implementing the appropriate interface. It will be asked by the transaction manger to vote on a commit whether it is able to roll forward or not.
- The *transaction client* (see section 3.6.4), which requests the server to create a new transaction, which it can use to perform operations on transaction participants and finally

---

[7] The term *participant* refers to *transaction participant* in this section

completes the transaction by requesting to commit or abort it.

A transaction is created upon a client's[8] request by a transaction manager. Since transactions are leased, the client has to provide a desired lease time (see chapter 3.5 for details about leases) on transaction creation. If the client does not *complete* the transaction before the lease expires, the transaction manager, which observes the transaction, will abort it. Apart from that a transaction is assumed to be *active* as long as the client does not request the manager to complete the transaction by requesting to either commit or abort the transaction, or the lease of the transaction expires.

The sequence diagram of the Jini transactions procedure is depicted in figure 3.12. It shows the afore-mentioned three parties (a single client, two participants and a transaction manager) which participate in a typical Jini transaction procedure that commits. The sequence is as follows:

1. The client has to find a Jini transaction manager in the Jini community. Therefore, it queries a lookup service to send it a reference to a transaction manager.
2. The client requests the transaction manager to create a new transaction that is leased for the desired period.
3. After receiving the new transaction from the transaction manager, the client can use it to perform operations on various services which support Jini transactions.
4. The service has to join the transaction as a participant, at the first time one of its transactional methods is called.
5. If the client has finished its work it will request the transaction manager to commit the transaction, which thereupon initiates the two-phase commit protocol by requesting all participants to "prepare" the commit of the transaction.
6. Participants return their decision if they are ready to roll forward in the next step to the transaction manager.
7. The transaction manager receives all participants' responses, decides to commit the transaction and notifies the participants to roll forward.
8. At last all participants commit the transaction making all changes under the transaction permanent.

Furthermore Jini transactions enable to group transactions together into a single transaction. Such transactions are called *nested transactions*, or also known as *subtransactions*. A nested transaction behaves as if it was a normal transaction, but it is wrapped inside a parent transaction. Aborting a nested transaction forces neither the parent nor other nested transactions of

---

[8] The term *client* refers to *transaction client* in this section

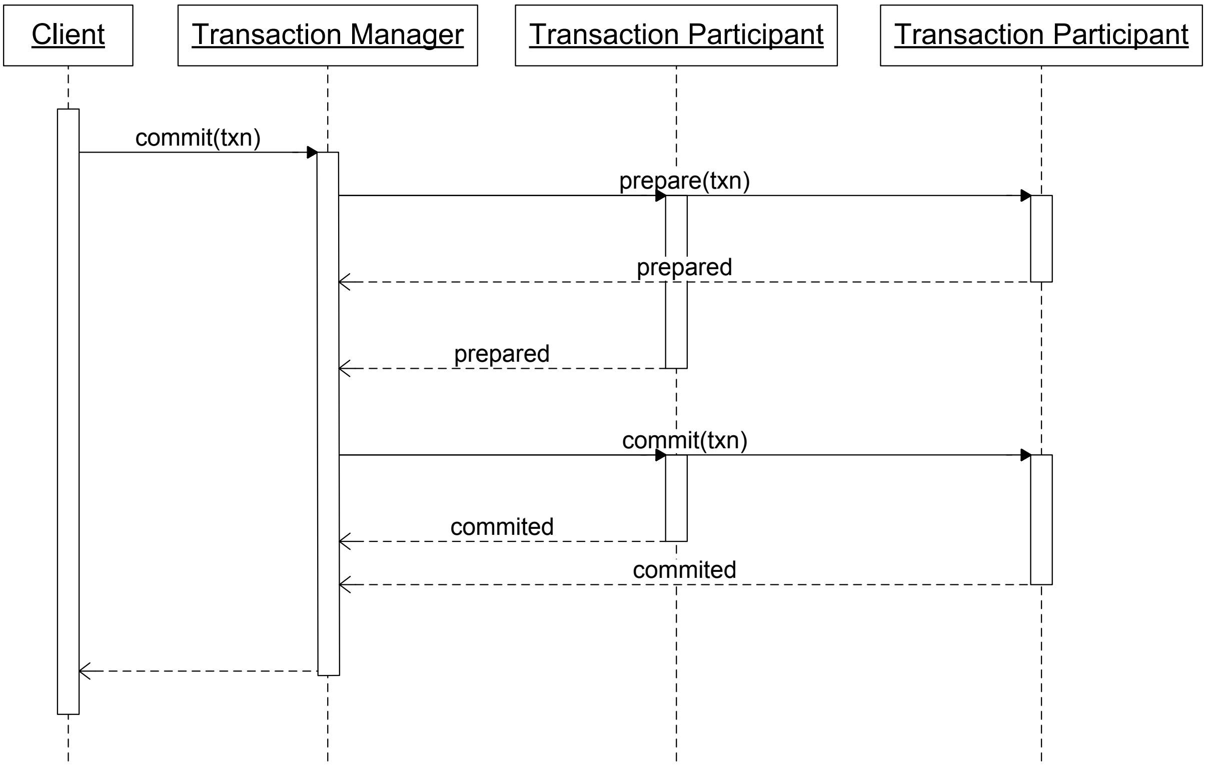


Figure 3.12: Sequence diagram of the two-Phase-Commit protocol

the parent to be aborted. Nested transactions are beyond the scope of this thesis and will not be discussed in detail. A description can be found in [53].

The Jini transaction specification is only a guideline enabling the use of transactions between Jini clients and services, the implementation details itself are left to the supporting Jini service and are not predetermined. In the following the specification of the two-phase commit protocol will be presented that has to be supported by a Jini service, which offers support for Jini transactions.

3.6.1 Two-Phase Commit Protocol

The two-phase commit protocol mainly describes how it can be guaranteed that a transaction, which may consist of distributed operations on various services, is either carried out by all participants or none of them upon the client's request to commit. The two-phase commit protocol ensures that services, which are utilizing Jini transactions, can comply with the ACID properties [43] much easier. Giving a brief summary, the ACID properties are:

Atomicity A transaction is atomic if and only if its contained operations are either all performed or none of them is.

Consistency It must be guaranteed that the state of a system is still consistent after a transaction has completed.

Isolation Transactions that are performed on the same system at the same time must not be aware of each other. This means that resources used by an operation of a transaction have to be locked and are therefore unavailable for other transactions until the owning transaction has completed.

Durability After a transaction has committed successfully its changes are made permanent, and will still exist after a system failure.

The compliance to the ACID properties of Jini transactions depends on the implementation of the supporting Jini services, but not on the transaction manager's implementation.
As the name of the protocol implies, the process of committing a transaction is divided into two phases:

1. **Prepare Phase:** The transaction manager asks all participants that joined the transaction to *prepare* for commit. A participant must decide whether it is able to roll forward or not in the next step and return its decision to the transaction manager. With its decision to be able to roll forward the participant assures that it will make all changes, made under this transaction, permanent, when asked to do so by the transaction manager.
   This phase is commonly called the "Prepare Phase" but also referred to as the "Vote Phase" in terms of Jini transactions, because the participants are *voting* by returning their preparedness to commit to the transaction manager that will if the transaction can be committed or not.
2. **Commit Phase:** The transaction manager sends the order to roll forward to the participating parties if and only if each participant has voted independently to commit the transaction. If only one participant voted that it is not able to commit in the next step, the transaction will be aborted and the transaction manager sends the order to abort the transaction to all participants.

3.6.2 Transaction Manager

The transaction manager is a Jini service that either implements the `TransactionManager` or the `NestableTransactionManager` interface. It hast to be registered at a lookup service like any other Jini service. In general it is concerned with the creation of transactions and the coordination of the two-phase commit protocol, on a client's request to commit a transaction, between the transaction participants and itself. A transaction manager that implements the `NestableTransactionManager` interface additionally supports nested transaction.

Further description on the internal behavior of the transaction manager, whether supporting nested transactions or not, are beyond the scope of this thesis and can be found in [71].

3.6.3 Transaction Participant

A Jini service that wants to provide operations that can be performed under transactions must implement the `TransactionParticipant` interface and define transactional methods for the client. The participant must join the transaction, after the client has invoked one of its transactional methods with a non-*null* transaction, by calling the transaction manager's join method providing the transaction and a proxy that implements the `TransactionParticipant` interface, otherwise the operation will not be performed under the transaction. The `TransactionParticipant` interface allows the transaction manager to communicate with and coordinate the transaction participant. The provided methods of the interface are shown in the class diagram in figure 3.13 and are briefly described below:

- **abort**: The transaction manager requests the transaction participant to undo any changes made under the transaction.
- **prepare**: The transaction manager asks the transaction participant if it is ready to roll forward in the next step. The transaction participant has three possible responses:
  - **NOTCHANGED**: The operations performed under the transaction had no effects on the participant's side. It does not have to wait for the transaction manager's answer to commit or abort the transaction.
  - **PREPARED**: The transaction participant is ready to make the changes permanent.
  - **ABORTED**: A failure occurred on the participant's side. Hence, the participant is not able to roll forward on a commit request.
- **commit**: The transaction manager informs the participant that it can roll forward now, making all changes under the transaction permanent.
- **prepareAndCommit**: If the transaction manager has "prepared" all participants and received only "NOTCHANGED" responses from all of them but one, it could call *prepareAndCommit* on the remaining participant. The semantic of this method must be the same as first to invoke *prepare* and then if it does not return *ABORTED* to call *commit*.

3.6.4 Transaction Client

In the Jini transaction procedure the client is the initiator. It requests the transaction manager to create a new transaction and uses the transaction to unite several operations on various services.

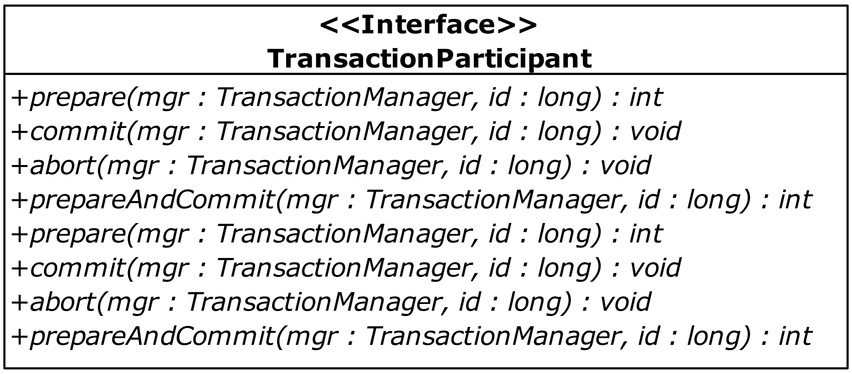

<<Interface>>
TransactionParticipant
+prepare(mgr : TransactionManager, id : long) : int
+commit(mgr : TransactionManager, id : long) : void
+abort(mgr : TransactionManager, id : long) : void
+prepareAndCommit(mgr : TransactionManager, id : long) : int
+prepare(mgr : TransactionManager, id : long) : int
+commit(mgr : TransactionManager, id : long) : void
+abort(mgr : TransactionManager, id : long) : void
+prepareAndCommit(mgr : TransactionManager, id : long) : int

Figure 3.13: Jini TransactionParticipant interface

The client completes the transaction by either requesting to commit or to abort the transaction. The client does not need to implement any interfaces to use transactions but it has to look for a transaction manager at a lookup service in order to receive a new transaction that it can use for its operations.

3.7 Distributed Events

Sometimes clients and services are waiting for a certain resource to become available, for this reason they can either use a blocking remote method of another service, where the wanted resource is expected to be, or poll the service periodically. However this leads to unnecessary network traffic or will/may even block the calling process for a very long time.

With the support of distributed events Jini allows clients or Jini services to become a *remote event listener* for events they are interested in. To become a listener the client or Jini services must register with the object, also called the “event source” that is able to generate the event of interest. If an event was triggered, the event source informs all registered listeners about the occurrence of the event by sending them a so called “notification”, that is a class which implements the `RemoteEvent` interface.

Due to the fact that events are propagated over the network via RMI, the delivery of the notification is unreliable because it might get lost on the way to the listener. There is no guarantee that all listeners will receive the notifications at the same time, that they reach each listener in the same order or that a client receives the notifications anyway. Hence, each event has a consecutive numbering so that the listener is able to reproduce the chronological order of the occurrence of a certain event. It is left to the implementation of the event source whether it retries, or how often it retries to notify a listener if it is unreachable. Depending on the implementation it can also happen that a notification a received more than once by a listener. For this

reason, notifications have a unique ID and a consecutive numbering.
Because a listener may be only interested in a notification within a specified time, Jini specifies that registrations must be leased because they are nothing else than resources, which are managed by a service. Leasing event listeners is done in the same way as leasing entries (see section 3.5).
This section will first introduce the RemoteEvent interface in sub-section 3.7.1, followed by an explanation how listeners can register to be notified in section 3.7.2 and finally how listeners are notified by an event source in section 3.7.3.

3.7.1 Remote Event

After an event is triggered, the event source has to notify the listeners, which have registered for the occurrence of this kind of event. Listeners can register for various types of events defined by the service which supports Jini distributed events. If a listener has registered for more than one event at the same event source, it must be able to distinguish events from each other, if it receives a notification. Therefore, the event source has to attach an instance of the `RemoteEvent` class as a parameter of the call. The `RemoteEvent` class, the class diagram (depicted in figure 3.14) provides additional information about the occurred event for the listener:

- **eventID**: A unique ID[9] identifying the event.
- **seqNum**: The actual event's sequence number at the time of event generation.
- **source**: A reference to the event source, which generated the event. For instance, the returned object might be a proxy that implements the remote interface of the source. But this is left open to the implementation.
- **handback**: The handback object serves as a "reminder" for the listener to be remembered why it has originally registered to notifications of such events.

3.7.2 Registration of a Remote Listener

The Jini framework does neither specify in which way a remote listener can register for an event of interest at the event source nor what types of events the event source can offer for clients to subscribe. However it specifies that the listener must provide a remote object on event registration that implements the `RemoteEventListener` interface, a handback object, and that the event source must lease the registration. It is left to the event source whether it

---

[9] The ID's are only unique at a single service that supports distributed events.

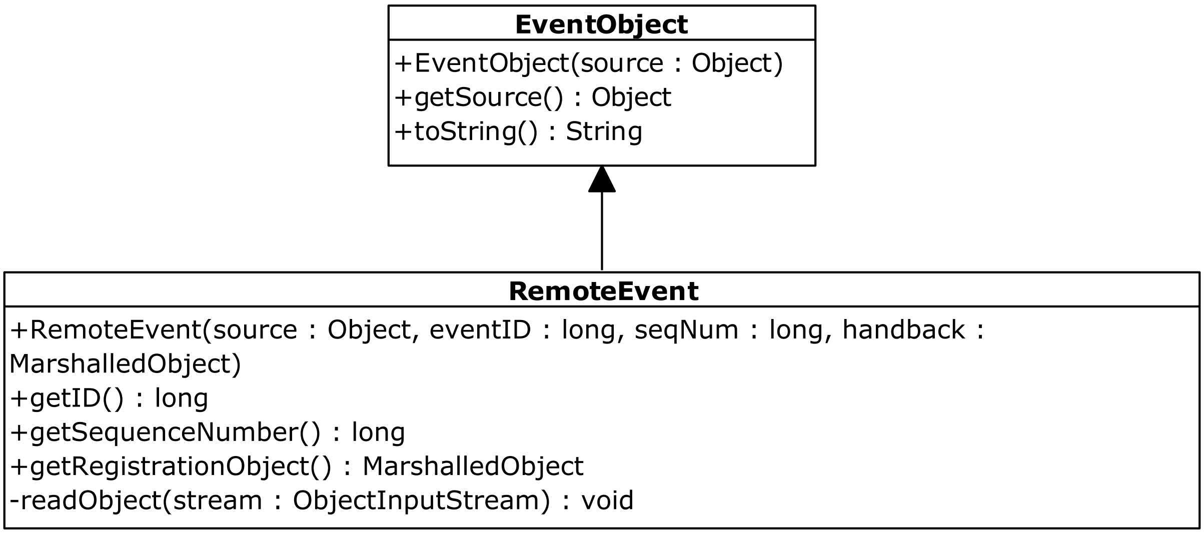

EventObject
+EventObject(source : Object)
+getSource() : Object
+toString() : String
RemoteEvent
+RemoteEvent(source : Object, eventID : long, seqNum : long, handback : MarshalledObject)
+getID() : long
+getSequenceNumber() : long
+getRegistrationObject() : MarshalledObject
-readObject(stream : ObjectInputStream) : void

Figure 3.14: Jini RemoteEvent class

prescribes how long a registration will be existent or it enables the listener to specify a desired lease time on registration. Furthermore the Jini distributed event specification recommends the event source to return an instance of the `EventRegistration` class (its class diagram is depicted in figure 3.15) as a confirmation that the listener was successfully registered. The `EventRegistration` class contains the following information:

- **eventID**: A unique ID identifying the event at an event source
- **seqNum**: The actual event's sequence number at the time of registration provided by the event source.
- **lease**: The lease object for the registration, containing how long the event source agrees to send notifications to the registered listener.
- **source**: A reference to the event source, which generated the event. For instance, the returned object might be a proxy or a remote interface of the source.

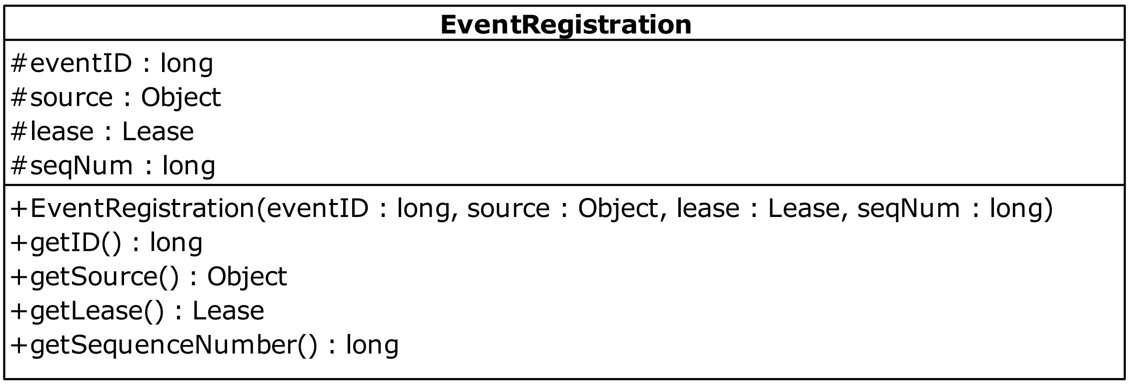


Figure 3.15: Jini EventRegistration class

3.7.3 Notification of a Remote Listener

As previously mentioned a listener has to provide a reference to its implementation of the `RemoteEventListener` interface when registration at an event source. The `RemoteEventListener` interface, the class diagram is shown in figure 3.16 specifies only the remote method *notify* which is used by the event source to inform the listener about the occurrence of the event of interest. On invocation the event source has to provide an instance of the `RemoteEvent` class to identify the occurred event. As already mentioned in section 3.7.1, the `RemoteEvent` class contains a unique ID, identifying the event itself, and a sequence number, which allows the listener to reconstruct the ordering of the events as they occurred at the event source. The sequence number is also helpful if the listener either receives the same notification more than once due to network problems or if the listener crashed and therefore missed notifications. In the first case the listener might drop the duplicate notification. In the second case it might request the service to resend the missed notification

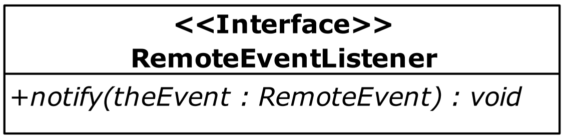


Figure 3.16: Jini RemoteEventListener interface

Chapter 4

JavaSpaces

The JavaSpaces [39, 23, 42] technology is a space based system developed by Sun Microsystems, based on the concepts of the Linda model developed by David Gelernter, which has been introduced in chapter 2. It enables Java distributed applications to collaborate, communicate and coordinate through Java object exchange using a shared memory between two applications instead of message passing or remote method invocation. The JavaSpaces technology is part of the Jini framework and is itself an API specification of a Jini service. As a Jini service, it makes use of other Jini services and components, like distributed transaction or events.

This chapter is dedicated to present an overview of the JavaSpaces service specification, starting with a description of the architecture (see section 4.1), followed by a comparison of the JavaSpaces technology and the Linda model (see section 4.2). Finally an insight into the usage of transactions and leases is given before explaining the JavaSpace and JavaSpace05 interface.

4.1 JavaSpaces Architecture

A JavaSpaces service is made up of objects, called "entries" that are placed into a persistent-, associative- and shared-memory, called "object space" or just "space"[1].

- The space is *persistent*, as entries placed into the space will remain there until they are explicitly removed from the space or their lease time expired (more about leases in section 4.4). Certainly the entries will be still lost, if services crash or shutdown and they do not use a persistent storage to store the entries.
- The space is *associative* because it finds entries by matching their type and content to a template rather than by their name or memory location. (templates and template matching were discussed in section 3.4.1)

---

[1] The term "space" refers to the implementation of the JavaSpaces service specification

- As a Jini service the space is *shared* because it is remotely accessible by several concurrently working processes.

The JavaSpaces service specification specifies a few simple operations (see the JavaSpace- and JavaSpaces05-API in section 4.6 and section 4.7.2), which enable to insert, read or remove entries from the space. These key features are the Java equivalents to those of the Linda model. A detailed comparison between JavaSpaces and the Linda model is given in chapter 4.2. The collaboration between processes using simple operations on various spaces is depicted in figure 4.1, taken from [39].

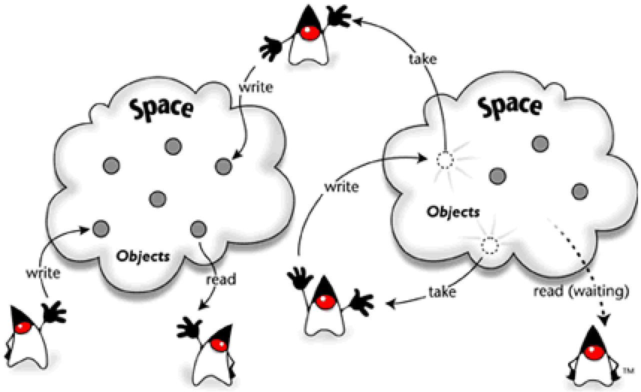


Figure 4.1: Collaborating processes across spaces.

As a Jini service the JavaSpaces service specification takes advantage of other Jini service and component specifications:

- Distributed Transactions: JavaSpaces uses the Jini transaction specification to bundle multiple operations, which are possibly distributed on various JavaSpaces, into a single atomic transaction. More about how JavaSpaces utilizes transactions in its service is described in chapter 4.3.
- Distributed Events: To avoid that clients block on a read or take request, until a specify entry is available, JavaSpaces offers notifications. Notifications build on the Jini distributed events specification. More about JavaSpaces notifications can be found in chapter 4.5 and the methods a client can use for registration are explained in chapter 4.6 and in chapter 4.7.
- Distributed Leasing: A JavaSpaces service assigns each entry and each notification registration a lease time. This prevents that neither unneeded entries waste service resources nor that clients are notified any longer than required. The utilization of leases by JavaSpaces services is described in section 4.4.
- Entries and Templates: Entries are the objects that are stored by a JavaSpaces service in its space. In order to retrieve an entry, a template has to be used because stored entries are

non-accessible directly via a name or reference. Templates are used to find an appropriate entry in the space. The JavaSpaces operations are described in section 4.6 and in section 4.7.

4.2 JavaSpaces compared to the Linda model

The JavaSpaces service specification is strongly influenced by the Linda model (described in chapter 2).

The JavaSpaces specification as well as the Linda model specifies an associative, persistent and shared memory that allows clients to collaborate, communicate and coordinate with each other. The shared memory is called “Java Space” by JavaSpaces and “Tuple Space” by the Linda model. The data objects, which can be stored in the space, exhibit another similarity. A single data object is represented by a list of typed fields in the Linda model and is called a “Tuple”, whereas in JavaSpaces it is an object of type “Entry” with fields declared as public. The operations allow to place, read or retrieve data objects from the space and are similar in both the JavaSpaces and the Linda model. Placing a data object into the space is achieved by the *write* operation in JavaSpaces and the *out* operation in the Linda model. The only difference between these operations, besides their naming, is that JavaSpaces specifies a lease time for each written entry and provides support for using transactions. This guarantees that entries can be removed when their lease time expires if they have not been taken in the mean time by a process. The read and take operations of both, the JavaSpaces and the Linda model, require to specify a “template” on invocation. This template is then used to match an appropriate data object in the space. The process of template matching is identical for both except for matching the field’s type. In this case the Linda model returns true if and only if the field’s type of the data object, stored in the space, and the corresponding field’s type of the template are equivalent. In contrast, JavaSpaces allows matching subtypes of entries, which means that a match returns true if either both entry types are equal or if the template’s entry type is a supertype of the compared entry’s type in the space.

The Linda model provides additionally the *eval* operation, which allows evaluating expressions within the space by specifying a “live tuple”. The result of which is a tuple that is again stored in the space. As stated in the JavaSpaces specification [7], JavaSpaces does not provide this functionality because it would require additional security restrictions, such as execution of arbitrary code only within a controlled environment, with limited resources and operations, which is shielded from the host-system, to execute arbitrary computation on behalf of the client. Aside from that it is possible to emulate the *eval* operation by using threads and specially extended entries.

On the other hand the Linda model neither contains notifications nor defines transactions or leases, all three functionalities that are part of the JavaSpaces service specification.

The results of the comparison between the JavaSpaces service specification and the Linda model are summarized in table 4.1.

	Linda model	**JavaSpaces**
Associative, shared memory	Tuple Space	Java Space
Data object	Tuple	Entry
insertion	out(tuple)	write(entry)
read	rd(template)	read(template)
consuming read	in(template)	take(template)
Evaluation in space	eval()	not supported
Notification	not supported	notify()
Transaction	not supported	Jini Transaction

Table 4.1: A comparison between JavaSpaces and the Linda model

4.3 Transactions

The JavaSpaces services take advantage of the Jini distributed transaction, see chapter 3.6 for details, to group several operations possibly over several JavaSpaces services into a single atomic transaction. JavaSpaces services must join each transaction as a participant in order to be informed by the transaction manager if the transaction shall be committed or rolled back. It is optional to the JavaSpaces client to be a participant too. For the sake of completeness, joining a transaction would allow the client to observe the transaction and being involved in the execution of the two-phase commit protocol. Figure 4.2 shows the interaction between JavaSpaces services, clients and the transaction manager in a distributed transaction. Each participating JavaSpaces service has to guarantee that on a commit either all modifications under a transaction will be committed or none of them. Equally it has to guarantee that on a rollback all modifications under a transaction are rolled back.

The various operations of the JavaSpaces and JavaSpaces05 API can be either invoked with a null or a non-null transaction. Invocations with a null transaction will be handled by a JavaSpaces service as if the operation was within a committed transaction. The JavaSpaces and JavaSpaces05 API define additional behaviors on their operations if a non-*null* transaction is used. The different behaviors of an operation in- or outside a transaction is explained later in section 4.6 and 4.7, after the various operations have been discussed.

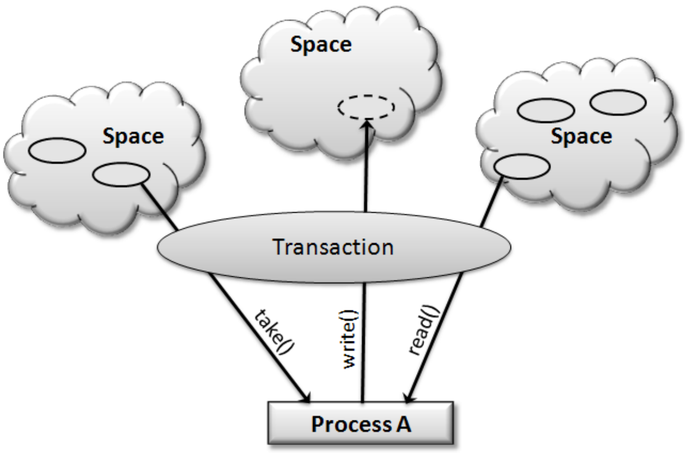


Figure 4.2: Operations across several spaces grouped into one transaction

4.4 Leases

JavaSpaces requires the definition of lease times for resources that are created on behalf of clients, such as entries and notifications. Leases shall guarantee that resources, which are no longer used by clients, do not waste the resources of JavaSpaces services unnecessarily.

On a successful write or notification's registration, clients receive a `Lease` object, which provides the operations *renew lease* and *cancel lease*. The *renew* operation allows the clients to further extend the lease of a resource before it expires, whereas the *cancel* operation can be used by the clients to announce that they are not further interested in the leased resource. The lease management in JavaSpaces builds upon the Jini distributed leases (see chapter 3.5 for details).

4.5 Notifications

JavaSpaces supports notifications in order that client processes are not blocked unnecessarily or have to poll the service until an appropriate entry is available in the space. Besides a RemoteEventListener, which will receive the notification from the service, the client must also specify a *handback* object (see section 3.7.1) provided in a marshalled form. Figure 4.3 shows a client (Process A) registering at a JavaSpaces service and later receiving a notification about a new entry inserted by another client (Process B).

JavaSpaces services utilize the Jini distributed events for notifications.

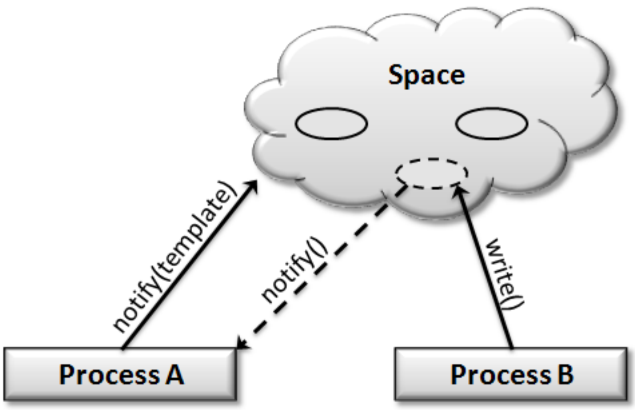
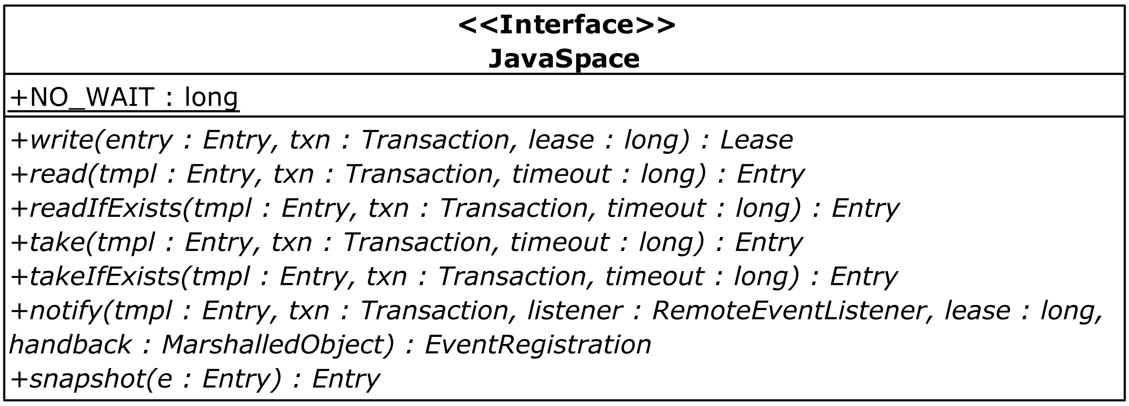


Figure 4.3: Client registers for notification

4.6 JavaSpace Interface

The JavaSpace interface provides the basic Linda model functionalities to JavaSpaces' clients. Either the JavaSpace interface or its extension JavaSpace05 interface (see section 4.7) must be implemented by a JavaSpace service. JavaSpaces services must either implement the `JavaSpace` interface, or the the *JavaSpace05* interface (see section 4.7). Figure 4.4 shows the class diagram of the `JavaSpace` interface, with a brief description of the provided methods.



<<Interface>>
JavaSpace

+NO_WAIT : long

+write(entry : Entry, txn : Transaction, lease : long) : Lease
+read(tmpl : Entry, txn : Transaction, timeout : long) : Entry
+readIfExists(tmpl : Entry, txn : Transaction, timeout : long) : Entry
+take(tmpl : Entry, txn : Transaction, timeout : long) : Entry
+takeIfExists(tmpl : Entry, txn : Transaction, timeout : long) : Entry
+notify(tmpl : Entry, txn : Transaction, listener : RemoteEventListener, lease : long, handback : MarshalledObject) : EventRegistration
+snapshot(e : Entry) : Entry

Figure 4.4: JavaSpace interface

write The *write* operation places a copy of an entry into the space. The same entry can be used for multiple write operations, because each *write* operation places a new entry into the specified space. It is necessary to provide the *lease* duration for the entry. The lease for an entry is handled by the service's lease manager. A successful write returns a Lease object, containing the granted lease duration. If the requested lease cannot be granted or the write of the entry failed a *RemoteException* is thrown.

Transactional Behavior: An entry written within a transaction is only visible for operations that are using the same transaction. If the same entry is removed by a *take* operation within the transaction, it will not become visible outside the transaction even if the transaction successfully commits. The roll back of a transaction causes all written entries under the transaction to be discarded.

read & readIfExist Reading requires to provide a template that is used to find an appropriate entry in the space via template matching. As described in section 3.4.1 a template is nothing else than an entry that is used by query operations and that may contain fields with no value assigned called wildcards. As a special case it is possible to pass a null reference for the template, resulting in a return of any entry that is available and not locked by another transaction. On success, a *read* will return a reference to a copy of the matching entry. Otherwise, the *read* waits until an entry is available that matches the template. In order to prevent the operation from blocking no longer than necessary a timeout value can be provided. If no matching entry is available until the timeout expires, a *RemoteException* is thrown which indicates the timeout expiration.

In contrast to the regular *read*, a *readIfExist* does not wait until a matching entry is found, except if there is a matching entry available, but the entry's transactional state is not yet certain.

Transactional Behavior: Using a transaction the read operation can match entries that are either written under the same transaction or already committed in the space. An entry that was read within a transaction that is still active, has the consequence that the entry is still visible to read operations but unavailable for take operations of other transactions. If a *readIfExist* finds an appropriate matching entry that is currently locked by a transaction, it will wait either until its timeout expires or the transactional state of the entry is certain.

take & takeIfExist The *take* behave in almost the same manner as the *read* operations, except that the matching entry is removed from the space.

On success, a *take* returns a non-null value (e.g. the entry) and the entry is removed from the space. If a take operation is executed within a transaction, the entry is locked and not available for other read operations of other transactions. It is not removed until the transaction has successfully committed.
Although a remote exception is thrown on failure, the entry might have been removed before the client has received it successfully, thus losing the entry in between. To prevent this from happening it is recommended to wrap the take operation inside a transaction, which is then not committed until the client has successfully received the entry.

Transactional Behavior: Entries removed within a transaction are no longer visible in the space. Neither a *read* nor a *take* operation will find the entries, even if they share the same transaction. A rollback of the transaction causes the entries to be restored and to become visible again.

notify In order that a process does not wait unnecessarily until a matching entry is available in the space, it can register its interest in future incoming entries. Blocking operations can be used as well, but are not appropriate since they block the entire application. Invoking the *notify* operation requires the specification of:

- A template specifying the entries of interest. The meaning of templates and template matching is identical to their meaning in *read* oeprations.
- The *RemoteEventListener* of the registering process, which will handle the notification.
- Upper bound of the lease time, which specifies how long the registering process is interested in receiving notifications about the registered event.

Transactional Behavior: Optionally a transaction can be provided upon the registration of a notification. In such a case the registered *RemoteEventListener* will receive notifications also about new matching entries, written under the same transaction, although the transaction has not committed so far. If no transaction was provided the listener will be notified only if matching entries are written under a *null* transaction or when a transaction has commited. Providing a transaction has effect on the lifetime of a registration, too. A registration is alive as long as the the lease or the used transaction are, meaning that a registration is removed when the provided transaction's state settles even, if the lease time has not yet expired.

snapshot It is highly probable that a template is used more than once for multiple *read*, *take* or *notify* operations. Because the process of serializing is expensive and the result of each serialization of the same entry is identical, the *snapshot* operation makes the work with the space more effizient. The *Snapshot* operation achieves this by creating a "snapshot" of the provided entry, which can be used by all operations on the same JavaSpaces service. The resulting snapshot is equivalent to the original entry as long as the original entry has not been modified. Modifications of the original entry have no effect on its snapshot.

4.7 JavaSpace05 Extensions

A drawback of the commonly used JavaSpace interface is that it does not provide methods for batch operations on the space, therefore the interface has been extended by the JavaSpace05[2] interface. The JavaSpace05 interface enables to use more than one entry or template on operation invocation and to retrieve more than one entry with a single query operation. Within the scope of the JavaSpace05 extension, the `MatchSet` interface was additionally added. First the

---

[2] The latest version of the JavaSpace05 interface is 2.1 released in 2005.

MatchSet interface and then the JavaSpace05 interface will be described briefly.

4.7.1 MatchSet Interface

The JavaSpace05 interface allows reading multiple entries with a single call. If the result is returned directly as a collection of entries to the caller, there is a risk that entries are removed from the space in the meantime, causing a wrong view of the space at the caller. The `MatchSet` interface has been introduced to circumvent this problem and to reduce resource consumption on the client side by managing the match set itself at the service side and by only exporting a proxy of an instance implementing the `MatchSet` interface. The interface (the class diagram is depicted in figure 4.5) allows the client to read incrementally the entries from the match set. The match set has a lease time, just as entries and notifications have, provided by the client and indicating how long it is interested in it. The lease time also guarantees that only those entries with a lease time equal or greater than the lease time of the match set are contained in the match set. This prevents that the MatchSet is invalidated for the reason that an entry is removed caused by an expired lease. If an entry is removed from the space by a take operation, the match set containing the entry must be invalidated by the service in order to ensure a consistent view of the space.

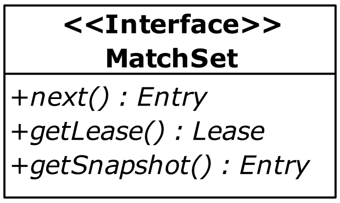


Figure 4.5: MatchSet interface

4.7.2 JavaSpace05 Interface

Figure 4.6 shows the class diagram of the JavaSpace05 interface.
The transactional behavior of the contained operations is equivalent to the transactional behavior of their corresponding singleton operations in the JavaSpace interface. In the following the methods of the JavaSpace05 interface will be described briefly:

write The *JavaSpace05.write* operation is an overload of the *JavaSpace.write* method allowing to write multiple entries with a single invocation to the space. The call of this method with a list of entries has the same effect on the space as if each of the entries was written

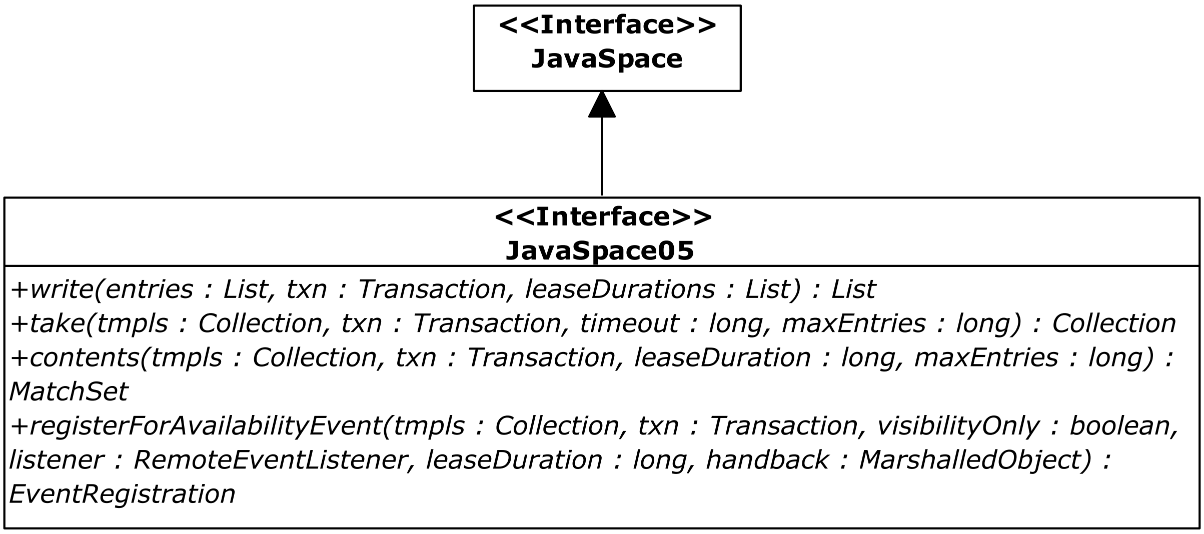

<<Interface>>
JavaSpace
<<Interface>>
JavaSpace05
+write(entries : List, txn : Transaction, leaseDurations : List) : List
+take(tmpls : Collection, txn : Transaction, timeout : long, maxEntries : long) : Collection
+contents(tmpls : Collection, txn : Transaction, leaseDuration : long, maxEntries : long) : MatchSet
+registerForAvailabilityEvent(tmpls : Collection, txn : Transaction, visibilityOnly : boolean, listener : RemoteEventListener, leaseDuration : long, handback : MarshalledObject) : EventRegistration

Figure 4.6: JavaSpace05 interface

by means of the *JavaSpace.write* method combined under one transaction. Although an ordered list is handed over on method invocation, it is neither guaranteed that the entries are stored in the same order as they occur in the list, nor that the entries will become visible to different observers in the same order.

contents This method is the counterpart of the singleton *JavaSpace.read* method, it allows to read multiple entries with more than one template. The found entries will match one or more of the provided templates. The results are stored in a match set, a remotely incrementally, iterable list, introduced in section 4.7.1. An upper bound, which is limiting the number of entries to be read, has to be specified on method invocation. The method will block until at least one matching entry is available.

take The *JavaSpace05.take* operation is an overload of the *JavaSpace.take* method allowing to retrieve more than one entry with multiple templates with a single method invocation. The fetched entries will match one or more of the provided templates. An upper bound, which is limiting the number of entries to be removed, has to be specified on method invocation. The method will block at last until one matching entry is available.
The invocation of this method with a list of templates has the same effect as fetching all available matching entries for each template with the singleton *takeIfExist* method combined under a single transaction.

registerForAvailabilityEvent Just like the singleton *JavaSpace.notify* method, this method is designed to notify a registered `RemoteEventListener` if a matching entry is available in the space. In addition it allows assigning a set of templates to a single event, causing the same event to be triggered by one or more templates. To prevent a notification flood, only one notification is sent for a single event, regardless of how many templates are matching the entry. In contrast to the *JavaSpace.notify* method, this method distinguishes between two kinds of transitions that an entry can make:

- From invisible to visible: An entry is assumed to be invisible if it is either not present in the space or if it was removed by a *take* operation within an uncommitted transaction.
- From unavailable to available: An entry is unavailable if it was read under a transaction and can not be removed by a *take* operation until the transaction has terminated.

The transition from invisible to visible is the same as the transition from unavailable to available. Table 4.2 shows the relation between the two transition types and the operations *read* and *take*. The leftmost column shows the two operations *read* and *take*, whereas the topmost row shows the two transitions invisible to visible and unavailable to available. The state after a transition, where an entry can be read or taken, is marked by a ✓.

	invisible ⇒ visible	**unavailable ⇒ available**
read	✓	✓
take	—	✓

Table 4.2: Relation of transition types and operations

On method invocation the caller can choose whether it wants to be notified on visibility only transitions or on both transition types. The RemoteEventListener receives an instance of the *AvailabilityEvent* class, a subclass of the *RemoteEvent* interface, which allows checking whether the event was triggered by a visibility or availability transition. Apart from that the *registerForAvailabilityEvent* method is similar to the *JavaSpace.notify* method. On method invocation the lease time and a transaction, which may be null, are required. The registration is removed as soon as the lease time has expired or if a non-*null* transaction was used and the transaction has completed. The application of transactions at the registration of notifications results in additional notifications that are triggered by activities within the same transaction.

Chapter 5

MozartSpaces

The Space Based Computing Group of the Institute of Computer Languages at the Vienna University of Technology has taken up the concepts of the Linda coordination model, introduced in chapter 2, in their development of a space based middleware: The eXtensible Virtual Shared Memory (XVSM) technology [37]. In contrast to other space based architectures like JavaSpaces, introduced in chapter 4, the XVSM technology is based on the concepts of a distributed, peer-to-peer (P2P) space whereas JavaSpaces is based on a centralized space architecture. XVSM' integrates the concepts of P2P-networks [20] to set up a distributed shared space amongst the participating and distributed clients, referred to as *peers*. XVSM is not restricted to a single space it enables the use of several spaces. Centralized space systems are rather prone to the failure of the space server or a network fault. In XVSM, if a peer becomes unavailable for any reason it does not imply that objects become unavailable. XVSM peers can use the space to coordinate, collaborate and communicate with each other, as it is the case with other space based systems.

In XVSM, the space is structured into containers whith operations that conform to those of the Linda coordination model. The container itself is only responsible to store objects and does not implement any coordination model itself. Hence so called *coordinators* take care of the coordination and administration of the stored objects. It is possible to assign several coordinators to a container and thus enabling the container to support various types of coordination mechanisms. The RandomCoordinator (see section 5.4) is attached to every container by default, enabling a random access to the stored objects. In order to insert or read an object some coordinators require the use of so called *selectors*, which provide additional coordinator specific meta-information that is required by the coordinator to store or find an object. Each selector is assigned to a coordinator, because the additionally contained information can only be interpreted correctly by the appropriate coordinator. XVSM distinguishes between two types of objects that can be stored in a container: *entries* and *tuples* (see section 5.2).

XVSM also supports the use of transactions (see section 5.6). XVSM is called "extensible"

because it allows extending and modifying the behavior of a container with the use of *aspects* (see section 5.8). Furthermore, XVSM provides notifications (see section 5.7) that are realized using aspects.
The Java based open source implementation of the XVSM technology, also developed by the Space Based Computing Group, is called “MozartSpaces” [62, 65]. Since it takes a central role in this thesis this chapter is dedicated to describe MozartSpaces in more detail. Starting with section 5.1 the MozartSpaces-API will be introduced before a brief description of entries and tuples is presented in section 5.2. Next, after a short description about containers in section 5.3, the coordinators that are already supported by MozartSpaces will be described in section 5.4, followed by section 5.5 that describes the corresponding selectors. The handling of transactions is explained in section 5.6 followed by section 5.8 about extending and modifying the behavior of containers by using aspects.

5.1 MozartSpaces-API

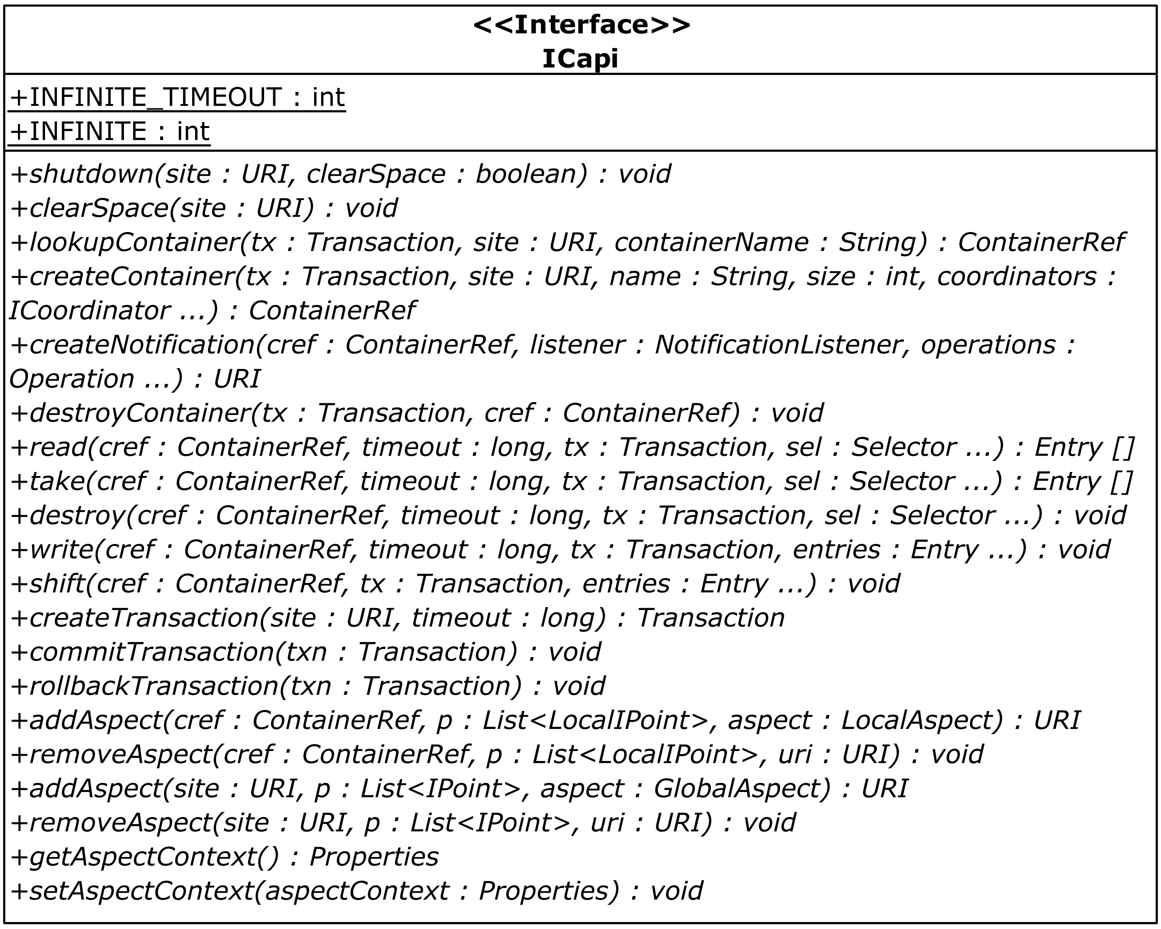
The MozartSpaces-API provides a set of operations which allow accessing the space. These are specified by the ICapi interface and can be categorized into operations that are either performed on the space itself or performed on a container. Figure 5.1 shows the class diagram of the ICapi interface.

The methods specified by the ICapi interface will now be described briefly. To give a better overview they have been categorized into the above mentioned two categories.

- **Operations performed on the space itself**
  Operations that are performed on the space itself, can be executed on the embedded or a remote space[1]. It is also possible to execute these operations under transactions.
  - **lookupContainer**: The operation lookupContainer can be used to retrieve a container reference for a given name, that already exists, from the space.
  - **createContainer**: Creates a new container, by providing the container’s name, the coordinators to be used and its maximum size. The container’s name must be unique in the space. The container’s maximum size specifies the maximal number of entries it is able to store. The caller receives a container reference if the operation completed successfully.
  - **destroyContainer**: Removes a container from the space, specified by the provided container’s reference.

---

[1] An embedded space runs in the same Java Virtual Machine (JVM) as the client that accesses it, whereas a remote space is running in a different JVM



```
<<Interface>>
ICapi
+INFINITE_TIMEOUT : int
+INFINITE : int
+shutdown(site : URI, clearSpace : boolean) : void
+clearSpace(site : URI) : void
+lookupContainer(tx : Transaction, site : URI, containerName : String) : ContainerRef
+createContainer(tx : Transaction, site : URI, name : String, size : int, coordinators :
ICoordinator ...) : ContainerRef
+createNotification(cref : ContainerRef, listener : NotificationListener, operations :
Operation ...) : URI
+destroyContainer(tx : Transaction, cref : ContainerRef) : void
+read(cref : ContainerRef, timeout : long, tx : Transaction, sel : Selector ...) : Entry []
+take(cref : ContainerRef, timeout : long, tx : Transaction, sel : Selector ...) : Entry []
+destroy(cref : ContainerRef, timeout : long, tx : Transaction, sel : Selector ...) : void
+write(cref : ContainerRef, timeout : long, tx : Transaction, entries : Entry ...) : void
+shift(cref : ContainerRef, tx : Transaction, entries : Entry ...) : void
+createTransaction(site : URI, timeout : long) : Transaction
+commitTransaction(txn : Transaction) : void
+rollbackTransaction(txn : Transaction) : void
+addAspect(cref : ContainerRef, p : List<LocalIPoint>, aspect : LocalAspect) : URI
+removeAspect(cref : ContainerRef, p : List<LocalIPoint>, uri : URI) : void
+addAspect(site : URI, p : List<IPoint>, aspect : GlobalAspect) : URI
+removeAspect(site : URI, p : List<IPoint>, uri : URI) : void
+getAspectContext() : Properties
+setAspectContext(aspectContext : Properties) : void
```

Figure 5.1: ICapi interface

- **createTransaction**: Creates a new transaction at the specified space.
- **commitTransaction**: Commits a transaction, making all changes performed under the transaction permanent, only if the operations within the transaction were successfully completed.
- **rollbackTransaction**: Rollbacks a transaction and discards all changes made within the transaction.
- **createNotification**: Creates a new notification on the specified container. This operation requires that the client specifies the condition when it wants to receive notifications and the listener to whom to send the notifications.
- **addAspect**: The addAspect operation is separated into a method that adds local aspects and into a method that adds global aspects to the space[2]. Both methods require providing the aspect itself and the IPoints[3], which specify where to attach the aspects. The local aspects additionally require specifying the container on which they shall do their work.

---

[2] The difference between local- and global-aspects is discussed in section 5.8

[3] IPoints are specified in section 5.8

  - **removeAspect**: The removeAspect operation is separated into a method that removes local aspects and into a method that removes global aspects from the space. Both methods require providing the IPoints where to detach the aspect. The local aspects additionally require specifying the container where they shall be removed.
  - **set-/getAspectContext**: These methods allow to set or query the properties of an aspect, e.g.: this can be useful to send authentication data to a security aspect.

- **Operations performed on a container**
  - **write**: Writes one or more entries into a container. Depending on the coordinator (see section 5.4) that was used at the time of container creation it is either required to use a selector, which provides additional information about how to store the entry, or not.
  - **read**: Reading one or more entries from a container always needs to make use of a selector. Selectors contain meta information for querying purposes, which are used to e.g. specify the number of entries to be read from the container, or which contain keys or represent templates.
  - **take**: The *take* operation works just as the *read* operation except that it also removes the entries from the container.
  - **shift**: The *shift* operation works just as the write operation except that if either the container is full or e.g. a specific key already exists, the entries are nevertheless written, causing the container to “shift out” entries. The selection of the entries that are removed depends on the coordinator’s behavior. It also might happen that the coordinator does not allow the shifting of any entry. In contrast to the other operations, *shift* is a non-blocking operation.
  - **destroy**: Destroy removes entries from the container. It is necessary to use a selector to specify which entries are to be removed. Although its usage is similar to read or take operations, it does not return any entries to the client.

5.2 Entry

An entry is the basic entity which can be stored in a container. It is an abstract class, and possesses a unique identifier. There are currently two derivatives of the abstract entry class: AtomicEntry and Tuple as depicted in figure 5.2.

An AtomicEntry represents a single entry in a container that holds the stored data’s type and value.

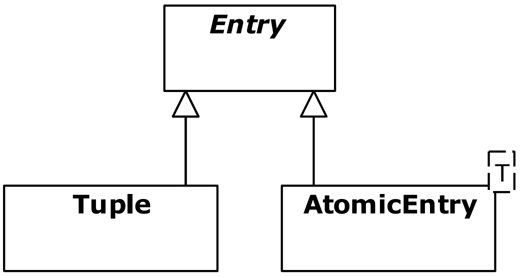


Figure 5.2: Entry types

A Tuple can be used to structure multiple entries and write them as one unit into a container. The number of entries a Tuple can contain is not restricted.
Accordingly a Tuple T can either hold:

- multiple AtomicEntries a_0, a_1, ..., a_n:

$$T = a_0, a_1, \ldots, a_n$$

- multiple Tuples t_0, t_1, ..., t_m:

$$T = t_0, t_1, \ldots, t_m$$

- a mixture of AtomicEntries a_0, a_1, ..., a_n and Tuples t_0, t_1, ..., t_m in arbitrary order. The numbers of AtomicEntries and Tuples do not necessarily have to be equal.

$$T = t_0, a_0, a_1, t_1, \ldots, t_m, \ldots, a_n$$

5.3 Container

MozartSpaces does not store data in a single storage. It rather makes use of several storages, called containers (e.g. spaces). A container is not just a simple storage. On creation it is possible to define the behavior of the container by specifying multiple *Coordinators* (see section 5.4). In addition every container owns a default coordinator: the RandomCoordinator (see section 5.4). Apart from specifying the container's coordination type, a container has a unique name and it is possible to limit the capacity of the coordinator, which is how many entries can be maximal stored by the container.

5.4 Coordinator

The coordinators are used to define how entries are written to or retrieved from a container. MozartSpaces distinguishes two categories of coordinators, the *implicit* and *explicit* coordinators. An implicit coordinator does not require additional information in case of writing an entry into the container. It is the coordinator's responsibility to correctly store the entry according to the coordinators' type. When reading an entry, it is again the coordinator's responsibility to retrieve the correct entry from the container according to the coordinators' type. In other words an implicit coordinator does the bookkeeping of an entry transparently to the user as defined by the coordinators' type.

In contrast, explicit coordinators require additional information if an entry is read or written. In this case the user is also responsible to contribute information for the administration of the entries in the container.

Listed below are the currently implemented coordinators of MozartSpaces divided into the above mentioned categories:

- **Implicit Coordinators**:
  - **FiFoCoordinator**: A Coordinator that organizes the stored entries in *first in, first out* manner commonly known as a queue. The entries are retrieved in the same order as they have been written to the container.
  - **LiFoCoordinator**: A Coordinator that organizes the stored entries in *last in, first out* manner commonly known as a stack. The entry that is retrieved is always the one that has been written last.
  - **LindaCoordinator**: The LindaCoordinator is an implementation of the Linda coordination model (see section 2) for MozartSpaces. Because the coordinator is based on the Linda model it is the only coordinator that restricts the type of entry it can store to the tuple class. To retrieve or destroy a tuple from the container, either with read, take or destroy, a template has to be provided. A template is a tuple that might contain fields whose value is *null*. Such fields are called wildcards, meaning that the value of the field is not of interest when looking for appropriate matching tuples in the container. To find matching tuples a template has to have the same structure, that is the same number of fields and ordering of the field types, as the Tuples that shall be found otherwise no matching Tuples can be found. The LindaCoordinator is like other tuple space implementations nondeterministic because it can store an unlimited amount of equal or similar Tuples; read, take or destroy cannot influence which tuple will be returned. It might happen that a tuple is never read, taken or destroyed.

    Although the LindaCoordinator requires a template when reading tuples it counts

    as an implicit coordinator because the user has neither an influence how tuples are stored nor on its nondeterministic behavior.
  - **RandomCoordinator**: As its name implies this coordinator offers only random access to its data structure. Because of that it is nondeterministic just like the LindaCoordinator. The RandomCoordinator is currently the *default* coordinator that is always added to a container on creation even if no coordinator is specified at container creation.

- **Explicit Coordinators**:
  - **KeyCoordinator**: Like a map the KeyCoordinator consists of a collection of unique keys and a collection of entries. Each key is assigned to one entry but its type is not confined to a specific type. When writing an entry to the container the key and the associated value have both to be provided. Later on the key is needed to successfully retrieve the entry from the container. It is not possible to retrieve an entry without knowing its associated unique key using the KeyCoordinator.
  - **VectorCoordinator**: The VectorCoordinator manages its stored entries in a modifiable list. This means that the position of the entries is not fixed. With this coordinator it is also possible to specify the position, where to insert an entry when writing it. If there is already an entry at the same position as where the new entry is requested to be written, the existing entry and all following entries are automatically moved up by one position. The entries are accessible by an index that is equal to the entry's position in the list. On retrieval it is possible to address each entry individually or to specify a subsequence of entries that should be retrieved from a particular index onward.

5.5 Selector

The selector [73] is used to supply the coordinator with additional information about an entry that shall be written to or retrieved from the container. All selectors have in common that it is possible to define how many entries shall be retrieved from or written to the container (representing the sementics of a bulk operation). Although implicit selectors can be omitted on writing and they do not provide extra information on retrieval, nevertheless they have to be provided in order to specify the coordinator that is used to store or retrieve entries. In contrast, explicit selectors are needed to provide additional information for the coordinator for both writing and retrieving of entries. There is one selector for each coordinator type (forming a selector - coordinator pair) and the implicit ones are only needed for retrieval. In section 5.3 it was mentioned that a container can own several coordinators. The selector, which is handed over on container access, specifies the coordinator that has to be used to proceed the request. Although the function of most selectors is obvious and self explanatory it is briefly described below, following the same categorization as the coordinators before.

- **Implicit Selectors**:
  - **FiFoSelector**: The FiFoSelector belongs to the FiFoCoordinator and hence it retrieves the entries in a first in, first out manner. Only the number of entries to retrieve can be specified.
  - **LiFoSelector**: The LiFoSelector belongs to the LiFoCoordinator and hence it retrieves the entries in a last in, first out manner. Only the number of entries to retrieve can be specified.
  - **RandomSelector**: The RandomSelector belongs to the RandomCoordinator and hence it retrieves the entries in a random order. Only the number of entries to retrieve can be specified.
    As is the case with the RandomCoordinator the RandomSelector is the base selector of a container, meaning that if no selector is used in a query the RandomSelector is used by default.
- **Explicit Selectors**:
  - **KeySelector**: The KeySelector belongs to the KeyCoordinator. On writing the entry and it's corresponding key have to be provided by the KeySelector to the KeyCoordinator. On retrieval the key has to be provided by the KeySelector to retrieve the designated entry from the container again.
  - **LindaSelector**: The LindaSelector belongs to the LindaCoordinator. In contrast to the other implicit selector - coordinator pairings, the LindaSelector is an explicit selector because it requires a template, which is then used by the LindaCoordinator, to retrieve matching tuples. On the other hand it is not necessary to specify the selector when writing a tuple to the container as it is the case with the implicit selectors.
  - **VectorSelector**: The VectorSelector belongs to the VectorCoordinator. On writing it is possible to specify the index, where to insert the entry in the list, besides the entry itself. On retrieval either only the index of the wanted entry or additionally also the number of entries to be removed starting at the given index is handed over.

5.6 Transactions

MozartSpaces transactions [65] are related to transactions in database management system. In MozartSpaces they are used to unite several space access operations on one or more containers into one atomic operation. As a matter of course the MozartSpaces transactions comply with the ACID properties [43] (which were already introduced in section 3.6) and are performed as pessimistic transactions, meaning that a transaction builds up locks either on the entries or possibly even on the whole container. This depends on the type of coordinator used by the container. MozartSpaces allows executing space access operations without specifying a transaction. In such a case MozartSpaces uses internally an implicit transaction, which is invisible to the executing client, to execute the operation and guarantee the compliance with the ACID properties. However if a transaction was specified by the client it corresponds to an explicit transaction in MozartSpaces. A MozartSpaces transaction can be committed on success, making all changes permanent or rolled back on failure, restoring the state before the changes.

5.7 Notifications

In order to minimize the number of blocking or polling operations MozartSpaces supports the concept of notification [62]. In contrast to the other features of MozartSpaces, such as transactions and the various coordination types, notifications are not part of the basis MozartSpaces API. They are realized by means of the coordinated usage of the appropriate aspects, which are provided by MozartSpaces itself.

The client registers itself for an event it is interested in at MozartSpaces and waits until it receives a notification from MozartSpaces meaning that the event has occurred. On registration for notification the client has to specify the container, the notification target it is interested in and a reference to the listener that should receive the notification. The notification targets currently available in MozartSpaces are:

- **read**: The listener will be notified when a read operation has been executed on the specified container.
- **write**: The listener will be notified when a write operation has been executed on the specified container.
- **take**: The listener will be notified when a take operation has been executed on the specified container.
- **shift**: The listener will be notified when a shift operation has been executed on the specified container.

- **destroy**: The listener will be notified when an entry has been removed as a result of an operation on the specified container.

The notification is sent to any client, which is registered, whether or not it was the initiator of the process, which triggered the notification event, on the container. The behavior of notifications according to operations within transactions is customizeable by means of the usage of the appropriate aspect.

5.8 Aspects

MozartSpaces allows adding additional functionality to the space or to a container by defining aspects [62]. An aspect defines an action that has to be performed either before an operation, called pre-aspect, or after an operation, called post-aspect, and are executed on the space or a container. MozartSpaces distinguishes two main categories of aspects:

- **Local Aspects**: Local aspects are limited to the operations performed on a specific container. They allow the definition of pre- and post-methods for the following operations that can be performed on a container:
  - addAspect()
  - removeAspect()
  - read()
  - destroy()
  - take()
  - write()
  - shift()

  To create a new local aspect only the abstract class `LocalAspect`, provided by the MozartSpaces API, has to be extended and the required pre- or post-methods overridden by the implementation.
- **Global Aspects**: As its name implies “Global Aspects” are executed globally, which is that they refer to the space itself or to all containers. They allow the definition of pre- and post-methods for the following operations that can be performed on a container:
  - createContainer()
  - destroyContainer()
  - transactionCreate()
  - transactionCommit()
  - transactionRollback()

  - coreShutdown()

  To create a new global aspect only the abstract class *GlobalAspect*, provided by the MozartSpaces API, has to be extended and the required pre- or post-methods implemented by the implementation.

Figure 5.3 shows how pre- and post-aspects are integrated into the control flow when using a local aspect on a container and a global aspect on the space.

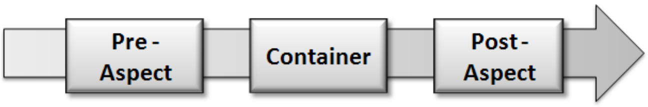


Figure 5.3: Control flow of LocalAspect on a container and GlobalAspect on space

After creating a new aspect it has to be integrated into MozartSpaces by specifying the point where the aspect's method has to be executed. Such a point is called *Interception Point* (IPoint) and is distinguished into the subcategories *LocalIPoint* and *GlobalIPoint*. These two categories are corresponding to the two basic aspect categories local- and global-aspect. It is possible to execute more than one aspect for the same pre- or post-method, but in such a case they are executed consecutively in the same order as they have been registered. Likewise it is possible to use an aspect more than once and in the case of local aspects for various containers too. Therefore the aspect needs only to be registered in MozartSpaces for the intended purposes as often as it is required.

Chapter 6

MozartSpaces LindaCoordinator Revised

The LindaCoordinator introduced in section 5.4 is a simple implementation of the Linda coordination model that has not been yet optimized. On closer examination of the Coordinator it turned out that a Java `HashMap` is used as its data structure. Although Java HashMaps provide a nearly constant-time performance for the basic operations, it turns out to be a bottleneck when one or more templates are used to find matching tuples. In this case it is inevitable to iterate over all tuples, contained in the HashMap, and match them against the given template(s). The larger the HashMap becomes the slower are appropriate tuples found via template matching.
The examination of the Coordinator also revealed that the kind of template matching that is specified by the JavaSpaces API is not supported. Concretely the existing matching algorithm does not support matching of subtypes even tough it supports type matching. Taken these deficiencies into account it turned out that a complete redesign of the Coordinator was more effective than adapting the original one. The term "LindaCoordinator" is referred to the revised coordinator that will be described and the original LindaCoordinator will be especially denoted as the "original LindaCoordinator" in this chapter.

This chapter deals with the revision of the LindaCoordinator in order to make it faster and provide a basis for the later implementation of a similar coordinator that is required by the XVSM JavaSpaces API in section 7. The redesign of the data structure is described in the first section 6.1, followed by the implementation details in section 6.2. The modification of the LindaCoordinator made it necessary to adapt the LindaSelector to comply with the new LindaCoordinator. The LindaSelector's modifications are described in section 6.3. Finally, section 6.4 presents the results of a benchmark comparison between the original LindaCoordinator and the revised one.

6.1 Redesign of the Data Structure

The redesign of the data structure required a detailed analysis of the tuple's structure and how template matching is performed. As explained, in section 5.2, tuples consist of multiple fields represented by entries. An entry holds a value and the type of the value. Template matching, which performs similar to the template matching of the Linda coordination model, uses a tupl (called template) that can contain "wildcards", which are not assigned field values, to find matching tuples in the data structure.

The first question that came up at the beginning of the design process was: "When are two tuples different from each other and therefore do not match?"

Two tuples $T = (F_T^0, F_T^1, \ldots, F_T^n)$ and $P = (F_P^0, F_P^1, \ldots, F_P^m)$, with field F^i containing a type and value pair, are different:

- if the tuple's number of entries are different: $|T| \neq |P|$
- if the corresponding field's types at position i differ: $F_T^i.type \neq F_P^i.type$ and $F_T^i.type$ is not a subtype of $F_P^i.type$
- if the corresponding field's values at position i differ: $F_T^i.value \neq F_P^i.value$

Aware of this fact the first data structure, which was selected, was a special tree structure, the *prefix trie* [38]. The trie specifies that strings sharing a common prefix are connected to a common node. This characteristic appeared to be useful for the problem. Instead of a string, the tuples' field order is used as prefix to index the tuples. This means that the path from the root node to one of the trie's leaf represents the size and the order of the tuple's fields. This approach is depicted in figure 6.1.

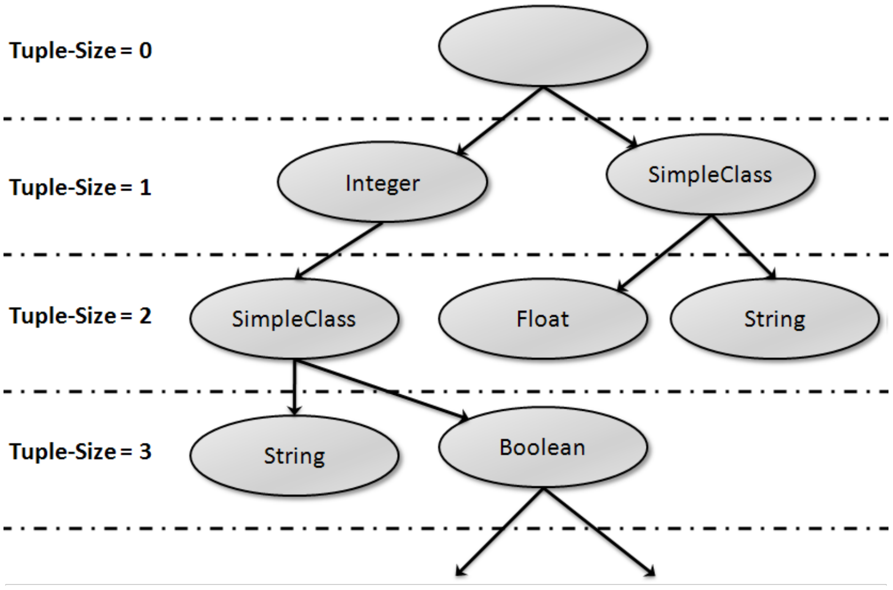


Figure 6.1: Trie as data structure

Considering the fact that it is more likely that tuples with the same size and ordering are inserted and retrieved from the container than tuples that differ in their size or ordering the trie approach is not optimal enough. On top of everything if only tuples with same size and ordering are stored, a single node would still contain all similar tuples in a single HashMap. As a result the performance would not be better than the performance of the original implementation.

Bearing this fact in mind and still holding on the tree structure the next approach would be to distinguish between different tuples as soon as possible. Therefore, the tree structure has been divided into several levels starting from the root node.

The first level, the "root-node" uses the tuple's size to make a first distinction between the tuples. The second level uses the field's ordering of the tuple to distinguish further between different tuples. The third level consists of a list representing the different fields of a tuple in the same order as they are arranged within the tuple. The fourth level maps the field's value to another list of tuples that share the same field's type and value at the same position. The resulting data structure is depicted in figure 6.2.

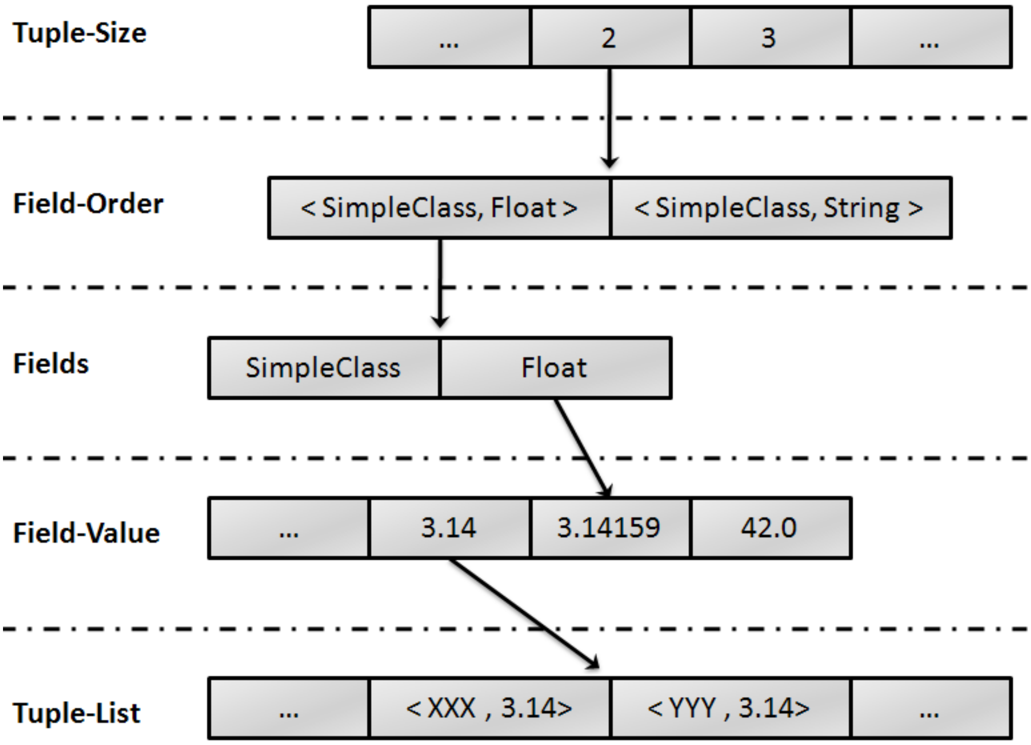


Figure 6.2: Tree Data-Structure including the different levels

6.2 Implementation

The RevisedLindaCoordinator is just like the original coordinator an implicit coordinator, since the user has no influence on how tuples are saved or retrieved from the container, and therefore derived from the `ImplicitCoordinator` class. In contrast to the existing coordinators the data structure has been separated from the coordinator class. This allows easier maintenance and increases replaceability of both components. Diagram 6.3 shows the architecture of the `RevisedLindaCoordinator` including the internal buildup of the `RevisedLindaStorage`, the class implementing the data structure. The RevisedLindaCoordinator conforms to the MozartSpaces container's specifications including the support of transactions.

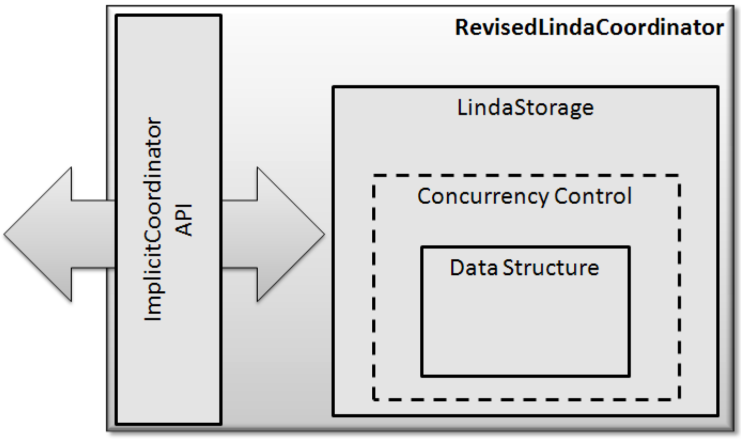


Figure 6.3: RevisedLindaCoordinator buildup

The `RevisedLindaStorage` class grants only indirect access to its internal data structure through the public methods shown in the class-diagram in figure 6.4. This is necessary in order to ensure that on the one side the data structure can only be modified through the predefined methods, which guarantee a consistent view for all concurrent accessing processes, and on the other side to enable the possibility to exchange the RevisedLindaStorage's data structure without affecting the remaining coordinator's implementation. Therefore, the synchronization between the concurrent processes is achieved by using the `ReentrantReadWriteLock` class to limit and coordinate concurrent read, write and delete access. The `ReentrantReadWriteLock` allows simultaneous reading by multiple readers as long as there are no writers. The write lock however is exclusive.

In comparison to the design, the implementation of the data structure forgoes to use a separate level for the tuple's size. Instead it has been combined with the tuple's field order into a single level, not only because they are correlative but also to reduce the number of maps used by the implementation.
The first level is represented by a `TreeMap`, using a so-called `TupleFingerPrint` class

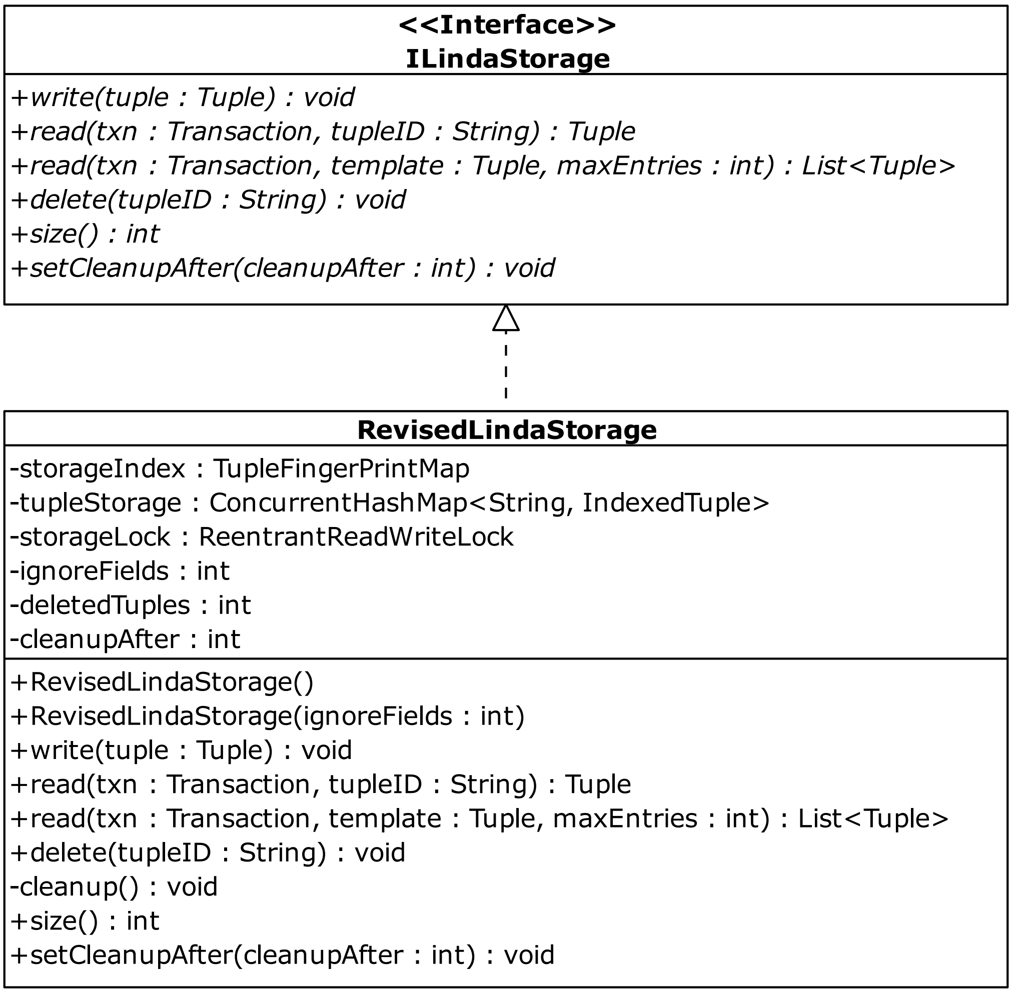

<<Interface>>
ILindaStorage
+write(tuple : Tuple) : void
+read(txn : Transaction, tupleID : String) : Tuple
+read(txn : Transaction, template : Tuple, maxEntries : int) : List<Tuple>
+delete(tupleID : String) : void
+size() : int
+setCleanupAfter(cleanupAfter : int) : void
RevisedLindaStorage
-storageIndex : TupleFingerPrintMap
-tupleStorage : ConcurrentHashMap<String, IndexedTuple>
-storageLock : ReentrantReadWriteLock
-ignoreFields : int
-deletedTuples : int
-cleanupAfter : int
+RevisedLindaStorage()
+RevisedLindaStorage(ignoreFields : int)
+write(tuple : Tuple) : void
+read(txn : Transaction, tupleID : String) : Tuple
+read(txn : Transaction, template : Tuple, maxEntries : int) : List<Tuple>
+delete(tupleID : String) : void
-cleanup() : void
+size() : int
+setCleanupAfter(cleanupAfter : int) : void

Figure 6.4: RevisedLindaStorage class

as the key's type and a HashMap for each assigned value representing the second level. The `TupleFingerPrint` class contains the tuple's size and the list of types in the same order as they appear in the tuple. The compare method of this class is responsible to compare the stored tuple's size and the order of the types to the ones contained by the passed instance of a `TupleFingerPrint` object. Furthermore, while comparing the internal type list to the passed `TupleFingerPrint`'s type list, it is verified if the passed `TupleFingerPrints` type at position i is derived from the type at position i of the internal type list. This has been done to provide support for the JavaSpaces API which requires subtype matching when reading or taking with a template.

The second level is represented by a list of `Concurrent-Hash-Maps`[1], which is ordered in the same way as the type list contained by the `TupleFingerPrint` one level higher. In this level it is enough to keep the ordering because the matching of the types was already done in the first level of the tree structure. The `Concurrent-Hash-Maps`, contained in the list of the second level, represent the third and last level. The field's value is used as the HashMap's key, which is then mapped to a list that contains references to the indexed tuples. The special case in which the field's value is unassigned, it is not possible to store the tuple's reference straightly in the `Concurrent-Hash-Map`, because this kind of HashMap does not allow to use *null* values as keys. As a way out, a new class called `ValueMap` was derived from Concurrent-Hash-Map, which provides an additional list to store reference of tuples if the field's value is unassigned. Figure 6.5 shows the `ValueMap`'s class diagram as it extends the `Concurrent-Hash-Maps` with its additional methods.

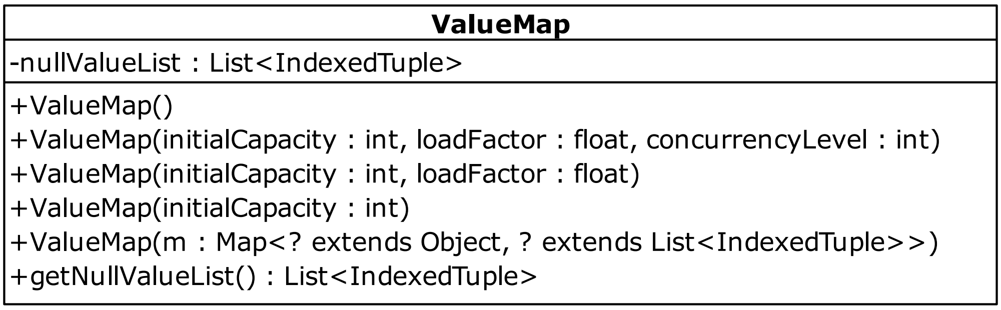

ValueMap
-nullValueList : List<IndexedTuple>
+ValueMap()
+ValueMap(initialCapacity : int, loadFactor : float, concurrencyLevel : int)
+ValueMap(initialCapacity : int, loadFactor : float)
+ValueMap(initialCapacity : int)
+ValueMap(m : Map<? extends Object, ? extends List<IndexedTuple>>)
+getNullValueList() : List<IndexedTuple>

Figure 6.5: ValueMap class

The final implemented data structure, inclusive the various components which have been used to realize the different levels, is depicted in the diagram in figure 6.6. Additionally to the data structure, the RevisedLindaStorage contains a normal HashMap, which assigns the tuple-ID to a so called `IndexedTuple`, a class that holds a reference to the tuple itself and to the lists of the leaves of the data structure. This enables faster access to the tuple or the referencing lists, if a specific tuple has to be looked up.

---

[1] Concurrent-Hash-Map is a HashMap that adds concurrent support.

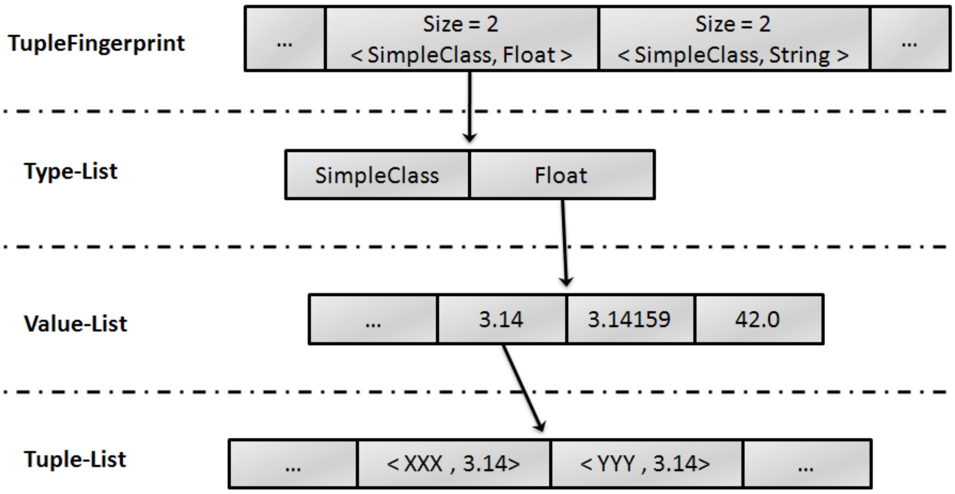


Figure 6.6: The data structure as it is finally implemented

For more flexibility, the RevisedLindaCoordinator has been extended by an optional parameter “ignoreFields”, which defines the number of fields to ignore at the beginning of each written tuple. These fields are ignored in a sense that they are neither involved in the way the tuples are indexed by the data structure nor that these fields are used for template matching. The parameter can be specified in one of the RevisedLindaCoordinator’s constructor.

6.2.1 Adding a Tuple to the Data-Structure

When writing a tuple, the root node is evaluated if the tuple’s `TupleFingerPrint` is already existent. If not, a complete new branch will be generated, right up to the tuple’s references in the leaves, and mapped to the corresponding `TupleFingerPrint` in the root node. Otherwise if there is already a mapping existent for the `TupleFingerPrint`, the corresponding subtree is examined if it contains a mapping for each field’s value and based on the result either a new mapping is added or the existing mapping is extended by a reference to the newly inserted tuple. At last a new tuple-ID to `IndexedTuple` assignment is added to the normal HashMap.

The current implementation of the RevisedLindaCoordinator has been optimized for fast tuple search and return, hence the complex process of tuple indexing slows down the speed of adding a tuple to the data structure.

6.2.2 Reading a Tuple from the Data-Structure

Reading is accomplished either by providing a string representing the tuple-ID to find a specific tuple or by using a template to find appropriate matching tuples. Former uses the normal

HashMap to find an appropriate mapping to the tuple-ID and either returns the tuple with the same tuple-ID or nothing.

If a template is used to find matching tuples, the template is first analyzed to determine the first field whose value is not *null*, to offer a speed up of the search process. After that the tree is descended up to the second level, where descending is continued along the path of the previously determined first none *null*-value field of the template. The remaining fields of the found tuples are compared to the remaining fields of the template. The resulting tuples are then returned to the calling process. If the template is composed of *null*-values only, the tree is searched through using the tuple's TupleFingerPrint and the first field's type as route. The resulting tuples, which are obtained by uniting the list of tuples contained in the located leaves, are then returned.

6.2.3 Removing a Tuple from the Data-Structure

Removing a tuple is only possible by providing the tuple's unique tuple-ID. The `IndexedTuple` that is assigned to the tuple-ID is looked up in the normal HashMap. The list of reference, contained in the `IndexedTuple`, to those leaves of the data structure, which contain a reference to the tuple itself, allows a fast removal of all tuple's references. But removing only the references instead of the whole branch if it is empty, results in a slow down of the data structure. To prevent this, the data structure has to be "cleaned up" after a certain number of deletions. The default value is set to 1000 removals, which has showed to be a good tradeoff, but the value is adjustable on coordinator instantiation.

6.3 Adapted LindaSelector

The original LindaSelector has been extended to allow the usage of a list of templates to find appropriate matching tuples. All of the tuples returned will match either one or more of the specified templates.

Due to the fact that neither the Linda model nor the original LindaCoordinator have a deterministic behavior, a new constructor has been added to the LindaSelector class which takes a tuple ID as a parameter. This allows the retrieval of a specific tuple from the container and on top of that the LindaCoordinator obtains a deterministic behavior.

These features are reflected by the newly added constructors and methods that are listed in class-diagram 6.7.

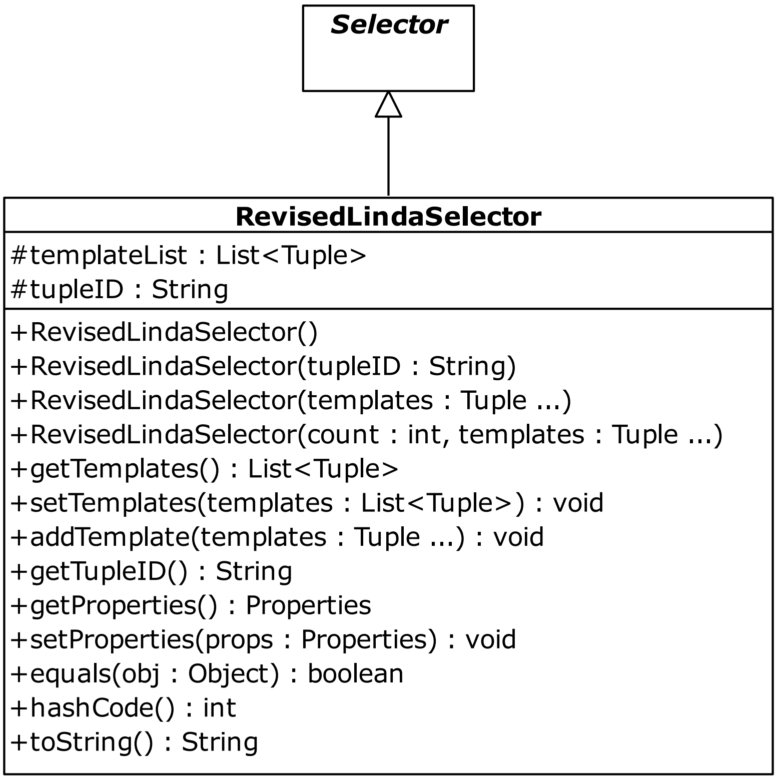

Selector
RevisedLindaSelector
#templateList : List<Tuple>
#tupleID : String
+RevisedLindaSelector()
+RevisedLindaSelector(tupleID : String)
+RevisedLindaSelector(templates : Tuple ...)
+RevisedLindaSelector(count : int, templates : Tuple ...)
+getTemplates() : List<Tuple>
+setTemplates(templates : List<Tuple>) : void
+addTemplate(templates : Tuple ...) : void
+getTupleID() : String
+getProperties() : Properties
+setProperties(props : Properties) : void
+equals(obj : Object) : boolean
+hashCode() : int
+toString() : String

Figure 6.7: LindaSelector class

6.4 Benchmark

In order to verify the increased performance of the revised LindaCoordinator, it has been compared to the original LindaCoordinator by executing several benchmarks. The benchmarks are divided into two categories: The first one measures the performance of both coordinators directly, whereas the second one analyzes the performance of both coordinators when used within the MozartSpaces system. Both categories have in common that the amount of time was measured each coordinator required to complete write, read and take operations. These operations were performed separately and independently from each other on both coordinators. Multiple measurements were made, each with a different number of tuples.

The tuples themselves are made up of 4 fields ordered in the same manner as listed in table 6.1. The table also shows beside the “template tuple”, two further versions of the tuple which differ only in their values. The “normal tuple” is written (n - 1) times, where n is the number of entries to be written within a measurement of the *write* operation. The “predefined tuple” is only written once and before the normal tuples by the *write* operation, because this special tuple is at the same time the one for which the read operation will look for during the measurement of the *read* operation. After the *write* operation’s measurement a total number of n tuples are contained in the coordinator.

Now that both coordinators contain n tuples, including the predefined tuple, the *read* operation's performance was measured. For this purpose the predefined tuple is required to measure the time how long each coordinator requires to find this special tuple out of the previously written n normal tuples.

In the final step the time is measured that each coordinator needs to perform as many *take* operations as entries contained by the coordinator. In order to achieve this, the "template tuple", depicted in table 6.1, is used to take all tuples that were previously written by the *write* operation measurement. All fields of the template tuple have a type but unassigned values, which allows to query all containing tuples one after the other out of the coordinator. The performance test of the *take* operation, executed on the coordinators directly, required the simulation of the behavior of the MozartSpaces system, because the coordinators do not support the retrieval and removal of a tuple in a single call. Therefore, a tuple was first read with the help of the "template tuple" and then using the returned tuple's ID, it was removed from the coordinator. Moreover, executing operations directly on the coordinators requires the simulation of transactions, because the coordinators are designed to perform all operations within transactions. Normally, if no transaction was specified by a client on operation's call, an implicit transaction is used by MozartSpaces to execute the client's operation. Apart from that, it was sufficient to use the operations provided by the coordinators and the MozartSpaces API. As a general restriction only a single tuple was written, read or taken at a time by each operation. All measurements were repeated 10 times, independently from each other, and the repetitions' mean value was used to finally generate the diagrams.

	Normal tuple	Predefined tuple	Template tuple
String	"Tuple"	"Benchmark"	null
Integer	[0 . . . (n - 2)]	42	null
Float	3,14159	3,14159265	null
Boolean	true	true	null

Table 6.1: Composition of the tuples

The relevant characteristics of the system, on which all benchmarks where performed, are listed in table 6.2.

Processor	AMD Athlon™ 64 X2 Dual Core 6400+ (each core with a frequency of 3.2GHz)
Main memory	2GB
OS	Ubuntu 8.04 32bit
Java™ Version	1.6.0_07-b06

Table 6.2: Computer's characteristics used for all benchmarks

Next the results of the benchmark will be discussed, starting with the results of the *write* performance measurement in section 6.4.1, followed by the results of the *read* performance measurement in section 6.4.2 and then by the results of the *take* performance measurement in section 6.4.3. At last an evaluation is given in section 6.4.4 about the performance of the revised LindaCoordinator compared to the performance of the original LindaCoordinator.

6.4.1 Write Performance Measurement

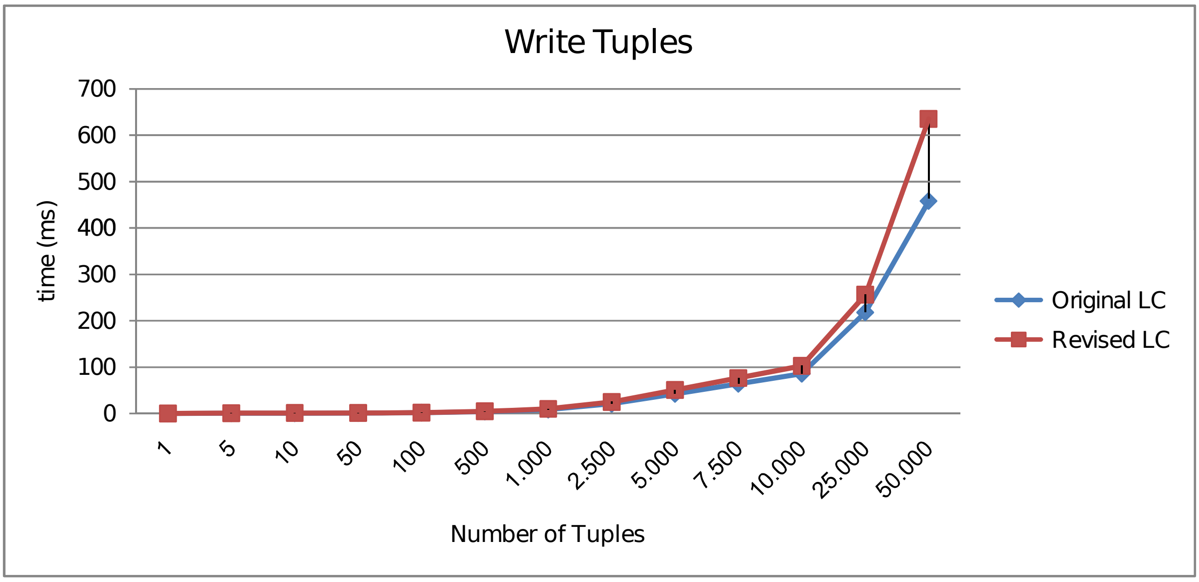
The results of the *write* operation's benchmark performed directly on the coordinators are presented in figure 6.8a. As it can be seen both coordinators consume nearly the same time up to 2500 tuples. After that it is clearly evident that the revised LindaCoordinator requires more time to store the tuples than the original LindaCoordinator. This is due to the overhead that is required by the revised LindaCoordinator's data structure to index the tuples.

The results of the *write* operation's benchmark performed via MozartSpaces on the coordinators are presented in figure 6.8b. In contrast to the benchmark performed directly on the coordinators, in this case both coordinators consume nearly the same time to write up to 10000 tuples. Beyond this number of tuples the revised LindaCoordinator is getting slower than the original one, but the difference between the measured times is lesser than the difference between the measured times of the coordinators when the benchmark was performed directly.

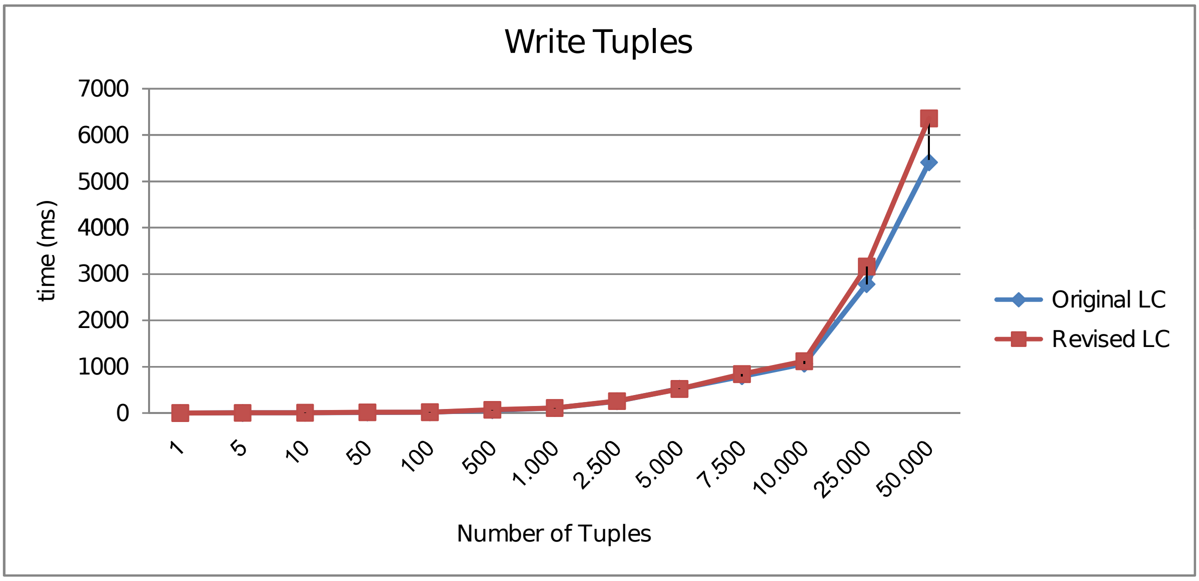
6.4.2 Read Performance Measurement

The results of the *read* operation's benchmark performed directly on the coordinators are presented in figure 6.9a. The revised LindaCoordinator requires a nearly constant time to find the predefined tuple independent from the number of tuples contained by the coordinator. On the other hand the original LindaCoordinator requires more time to find the predefined tuple out of 1000 and more tuples than the optimized coordinator. The original Coordinator must compare every single tuple to the given template to find appropriate matching tuples. This disadvantage becomes already noticeable at a number of only 1000 tuples, as it can be seen in the diagram.

The results of the *read* operation's benchmark performed via MozartSpaces on the coordinators are presented in figure 6.9b. The resulting diagram has nearly the same characteristics as the directly performed operation's diagram, but differs at a number of tuples less than 100. In this case both coordinators require more time to find the predefined tuple than finding it out of 500 stored tuples. As well the revised LindaCoordinator requires clearly more time to find the predefined tuple out of 50000 stored tuples in a read operation performed via MozartSpaces than finding it out of 25000. Comparing it with the results of the *read* operation's benchmark performed directly, these results are nearly constant only for a number of tuples between 100 and 25000.



(a) Performed directly on the coordinators



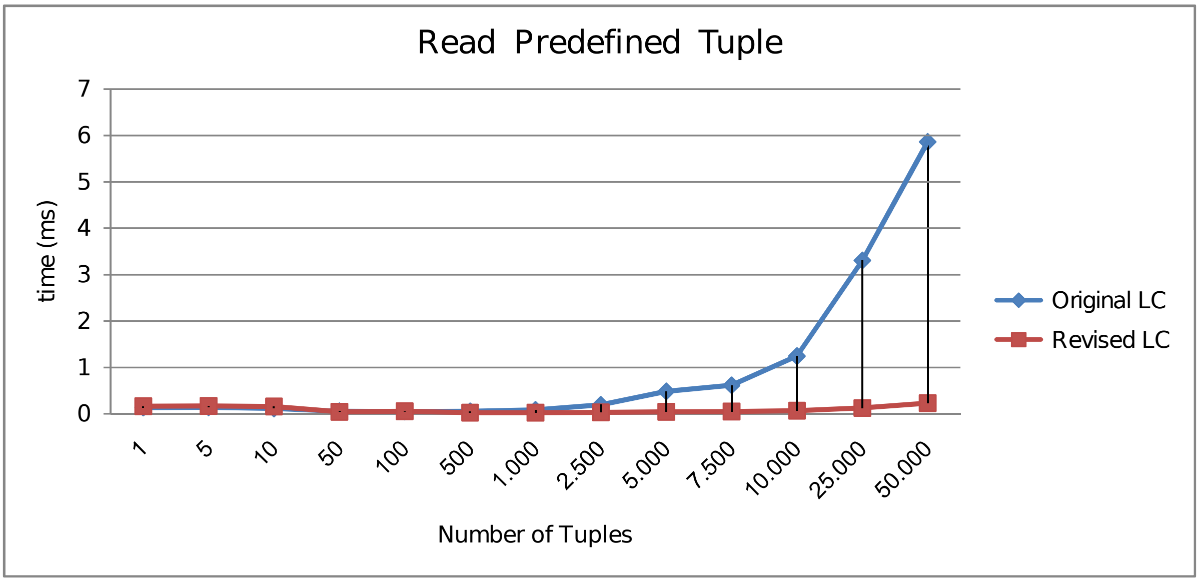
(b) Performed via MozartSpaces

Figure 6.8: Benchmark results of the “write” operation

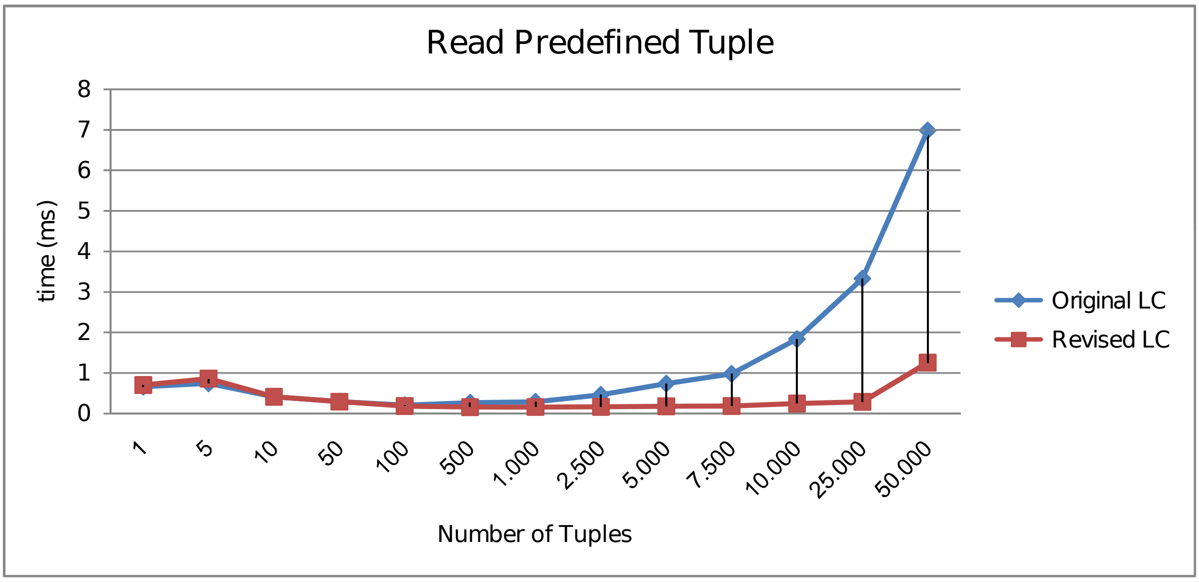
6.4.3 Take Performance Measurement

The results of the *read* operation’s benchmark performed directly on the coordinators are presented in figure 6.10a. Both coordinators are showing the same behavior up to a size of 10000 stored tuples, from this point on it is clearly that the revised LindaCoordinator is faster than the original LindaCoordinator thanks to its optimized data structure.

The results of the *read* operation’s benchmark performed via MozartSpaces on the coordinators are presented in figure 6.10b. It can be seen from the diagram that there are no major differences apparent up to a size of 10000 stored tuples. Both LindaCoordinator requires a little more time



(a) Performed directly on the coordinators



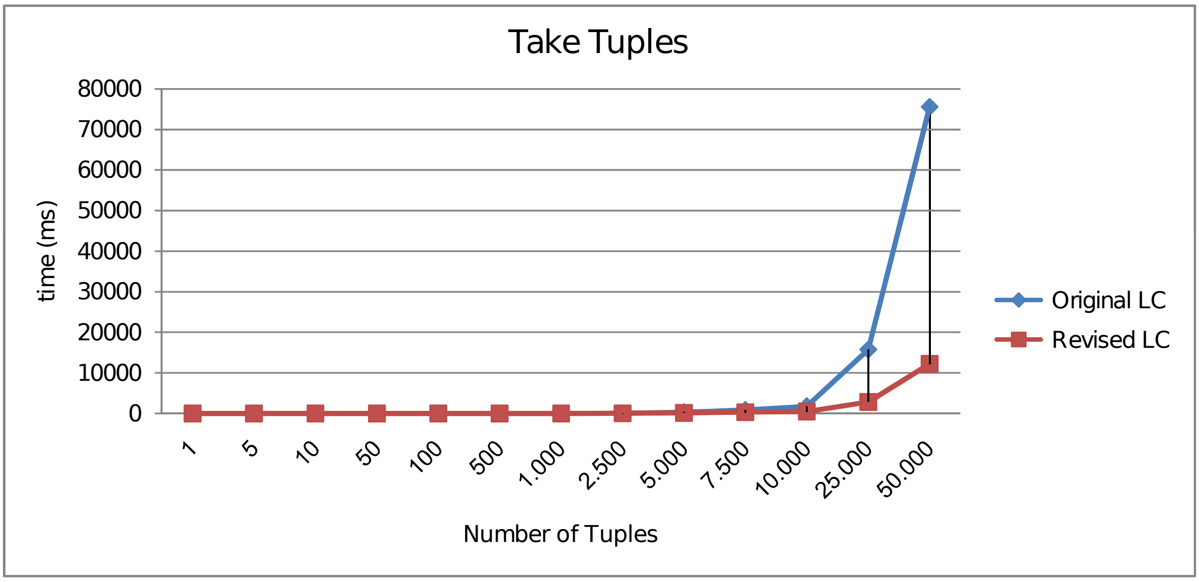
(b) Performed via MozartSpaces

Figure 6.9: Benchmark results of the “read” operation

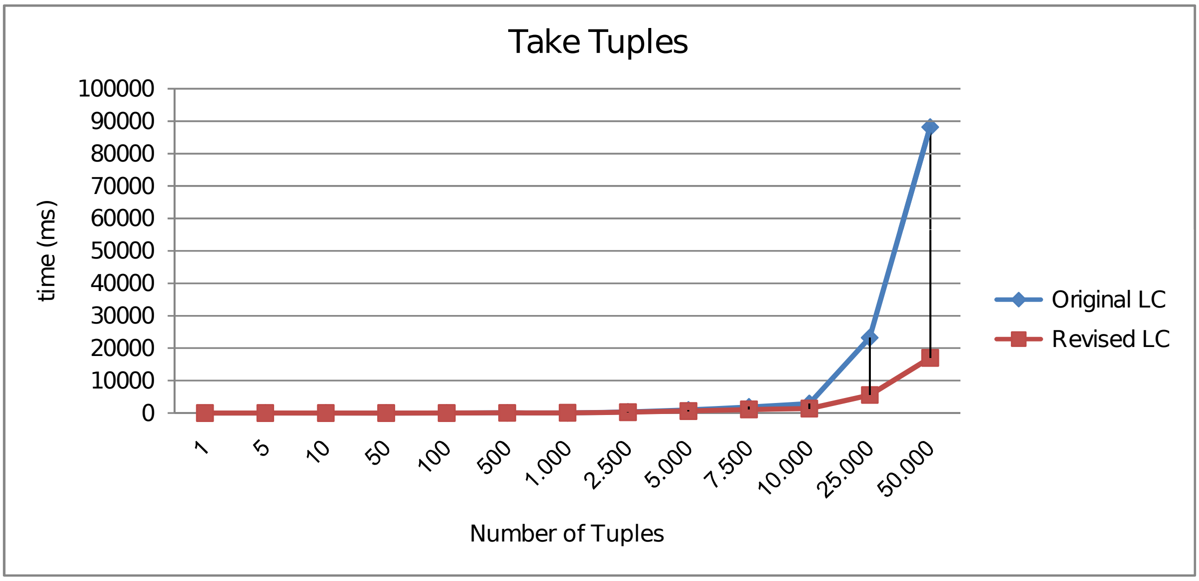
to take 25000 stored tuples, when performed via MozartSpaces as performed directly on the coordinators itself, and even doubles this time difference at a size of 50000 stored tuples.

6.4.4 Benchmark Evaluation

The benchmark results confirm the previously made assumption in section 6.2.1 that the revised LindaCoordinator is slower at *write* operations, but it is faster at *read* and *take* operations. As well both operations *read* and *take* have a constant time behavior up to a size of 10000 stored tuples, but are 2 times faster at a size of 25000 and even 4 times faster at a size of 50000 stored



(a) Performed directly on the coordinators



(b) Performed via MozartSpaces

Figure 6.10: Benchmark results of the “take” operation

tuples than the corresponding original LindaCoordinator’s operations. It is suppossed that Java internal mechanisms like re-organization mechanisms, e.g. hashmaps index restructuring, are responsible for the time consuming operation.

To conclude the revised LindaCoordinator performs better than the original LindaCoordinator at the operations *read* and *write* depending on the size of the stored tuples. This speedup comes with the disadvantage that the revised LindaCoordinator is slower in writing tuples than the original one. As already mentioned this is due to the data structure used by the revised LindaCoordinator that requires to index the stored tuple. One improvement would be to temporarily store the written tuple in a separated list and assign a thread with the task to index the tuple.

But in this case additional attention must be payed with read and take operations, because these operations will have to browse through the list of temporary tuples as well.

Chapter 7

The JavaSpaces API Standard for XVSM

The JavaSpaces API standard for XVSM (JAXS) enables existing JavaSpaces based systems to use MozartSpaces, the open source java implementation of XVSM, as an entry storage without the need of altering the existing system itself. Therefore it can also be seen as a “bridge” or “middleman” between the JavaSpaces API and the MozartSpaces API.

As introduced in chapter 4 the JavaSpaces API specification is a specification of a tuple space like Jini service that is built mainly on the following list of components offered by the Jini framework:

- Jini Entry as the objects to be stored in the space.
- Jini Transactions to combine various operations, possibly across various Jini services, into a single atomic operation.
- Jini Events for JavaSpace notifications.
- Jini Leases to prevent the waste of resources.

The main object of the implemented JAXS is to behave as a Jini service, which complies with the JavaSpaces API specification, to other services and clients in the Jini community and hide the fact that MozartSpaces is used as the back-end of the implementation. MozartSpaces not only provides the space where the entries are stored, it also takes care of other functionalities, like transactions, that are needed by the JAXS implementation. The JAXS service architecture will be described in more detail in section 7.1.

Although the JavaSpaces API is based on the concepts of the Linda coordination model, the way how stored entries are matched against a template in a *read* or *take* operation is different. The template matching, as described in section 3.4.1, specifies that a stored entry is a potential match against the template if it is a subtype of the used template. For this reason, the revised LindaCoordinator, which was implemented in the scope of this thesis and described in section 6, is unsuitable for the use as the storage of the JAXS, because it does not support this type of matching. The coordinator compares each field's type of a stored tuple to the corresponding field's type of the template. Since there is no simple way to constrain the type matching to specific fields in the revised LindaCoordinator, a slightly modified version, but based on the concepts of the revised LindaCoordinator, had to be written. The implementation details of the JSLindaCoordinator are presented in the section 7.2.

The JAXS is further responsible to take care of the interaction between the JavaSpaces API, which is used by its clients, and MozartSpaces, which is used in the background. It is responsible for:

- The conversion of Jini Entry into a storable MozartSpaces tuple.
- The management of Jini transactions and the mapping of those to MozartSpaces transactions.
- The management of Jini notifications.
- The management of Jini leases.

Jini entries must be converted into MozartSpaces tuples by the JAXS before they can be processed by the MozartSpaces operations. Certainly the tuples must be reconverted to Jini entries by the JAXS again before they are returned to the JAXS's clients. The conversion of Jini entries to storable MozartSpaces tuples and back again will be explained in section 7.3.

The JavaSpace API uses Jini transactions for its transactional operations, but these are incompatible with MozartSpaces transactions. Mapping Jini transactions to MozartSpaces transactions is possible but not simple. On the one hand, Jini transactions are based on a two-phase commit protocol that must be supported by services. On the other hand, services must join as transaction participants each transaction that is used by clients when invoking the services' operations. The management and the implementation for the support of Jini transactions will be described later in section 7.5.

JavaSpace notifications differ in their semantics from the notifications provided by MozartSpaces. A significant difference is that JavaSpace notifications allow the specification of a template on registration, which constrains the notification to certain matching entries, but MozartSpaces does not provide any equivalent mechanism for its notifications. Another difference is that JavaSpace notifications are bound to a lease that is granted on registration. In contrast, MozartSpaces notifications are bound to a number of times, specified at registration, that the registered listeners should be notified. For this reason it was essential to write a custom notification module JAXS, whose implementation is described later in section 7.6.

The JavaSpace specification defines the application of leases amongst other things for stored entries and notifications. On the basis of the fact that MozartSpaces does not provide a lease management or something comparable, it was essential to write a lease management component for the JAXS based on the Landlord framework (see section 3.5.2 for details). Section 7.7 is dedicated to the implemented lease management's description.

7.1 The JavaSpaces-MozartSpaces API Service Architecture

The JAXS's internal architecture is depicted in figure 7.1, which shows three main architecture components:

JAXS Service Proxy The JAXS service proxy is the front-end through that the JavaSpaces clients are able to communicate with the JAXS service. It is provided by the JAXS service during the registration process at a Lookup service and is forwarded by the Lookup service to the client on its service lookup request. The service proxy provides the JavaSpaces API to the clients that can use it to interact with the JAXS service. The proxy does not only forward the client's requests to the JAXS service and vice versa, but is also responsible to preprocess the data before delivering it either to the JAXS service or to the client. The preprocessing includes among other things the conversion and reconversion between Jini entries and MozartSpaces tuples.

JAXS Service The JAXS service is the core of the JAXS, the "bridge" between the JavaSpaces API and the MozartSpaces API. It is in charge of behaving as an ordinary JavaSpaces service to its clients that includes the support of Jini transactions, Jini Event notifications and leasing of resources. Most of these are implemented using the features provided by MozartSpaces, but there are features like leasing which have to be handled and managed by JAXS service.

MozartSpaces MozartSpaces and its functionalities are used as storage for the entries, as transaction handler for the JAXS operations and as a part of the notification system implemented by the JAXS.

How JAXS fits into a Jini federation is depicted in Figure 7.2. As a Jini service it is operating in the same manner as any other Jini or especially JavaSpaces service. The fact that another space based middleware is working in the background is hidden from the accessing Jini clients and services. They are using the proxy of JAXS service, which they can lookup at the Lookup service where JAXS is registered, to interact with the service. JAXS can wrap the operations executed by the clients or services into Jini transactions on their demand.

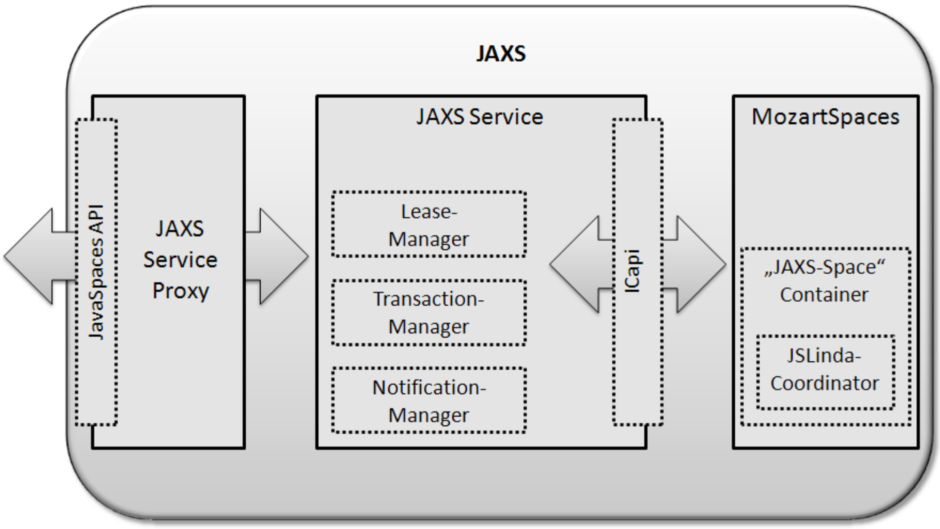


Figure 7.1: JAXS architecture

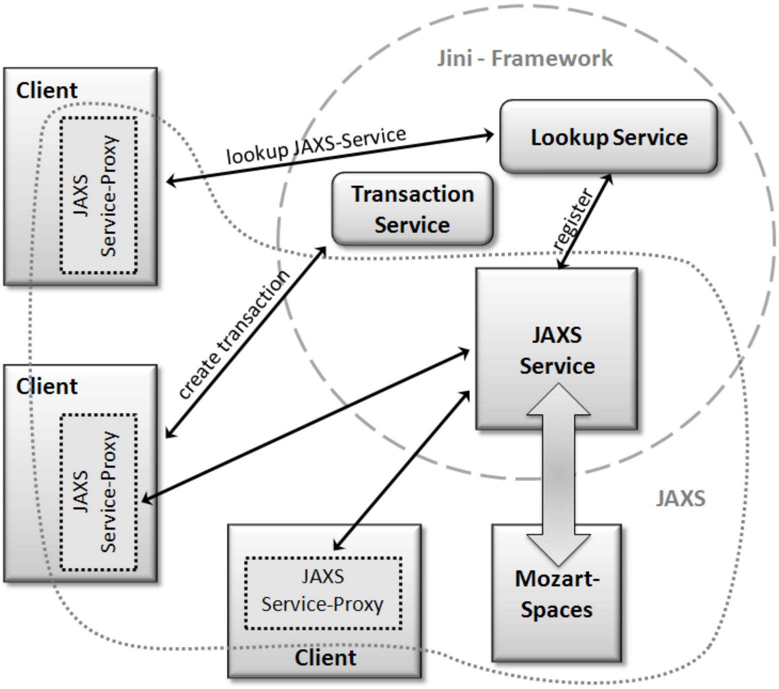


Figure 7.2: JAXS system integration

7.2 JSLindaCoordinator

The JSLindaCoordinator is a slightly modified version of the revised LindaCoordinator introduced in section 6. The only difference is that the JSLindaCoordinator meets the requirements of the Jini template matching specifications (see section 2.2.4) to support the subtype matching of Jini entries. The standard LindaCoordinator is not able to do this because it does subtype matching for every field of a tuple and not just for the entry's class. Although it would be possible to store the entries class type as one of the tuple's fields, template matching would not work either because the revised LindaCoordinator uses all tuple fields to generate a fingerprint of the tuple. This fingerprint is later used to find potential matching tuples against a given template in the first instance. If there are subtypes of a template available, they won't be identified as potential matching tuples if their number of fields differs from the template's field count. Due to of the tuple's fingerprint they are treated as different non matching tuples. For this reason the data structure, which is used by the revised LindaCoordinator, has been slightly modified to meet the JSCoordinator's requirements.

7.2.1 Implementation

As previously mentioned the only difference is how the TupleFingerPrint is generated and used to index the stored tuples. Therefore only this difference will be discussed in this chapter.
The JSLindaCoordinator's data structure allows only to store instances of `JSTuple` class, a subtype of the `Tuple` class. The composition of tuples, which can be stored, has been restricted to the subclass. The first field that is not ignored must provide the class type of the Jini Entry to be stored. As depicted in the class diagram in figure 7.3 the JSTuple has an additional field "jsEntryType" to store the class type of the original Jini `Entry` instance. The tuple's fingerprint is therefore defined by the provided class type and not by the types of all tuple fields. The remaining fields are then used as in the revised LindaCoordinator's data structure. They are stored in an ordered list, which is linked to the corresponding tuple's fingerprint, where each list's entry refers to a ValueMap, which is already known from the revised LindaCoordinator. The ValueMap map's the field's value to a list of references of stored tuples. The data structure that is used by the JSLindaCoordinator is depicted in figure 7.4.

The JSCoordinator is not restricted to the subtypes of Jini `Entry` classes and hence is usable for any other purpose that requires such a coordinator.

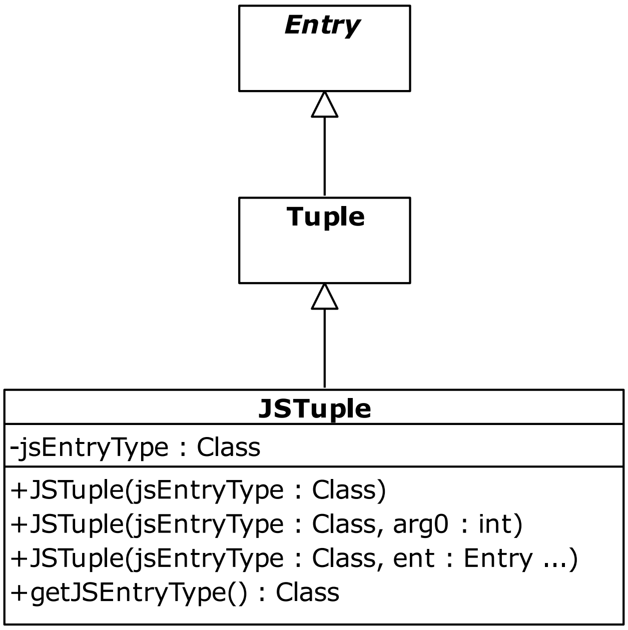

Entry
Tuple
JSTuple
-jsEntryType : Class
+JSTuple(jsEntryType : Class)
+JSTuple(jsEntryType : Class, arg0 : int)
+JSTuple(jsEntryType : Class, ent : Entry ...)
+getJSEntryType() : Class

Figure 7.3: JSTuple class

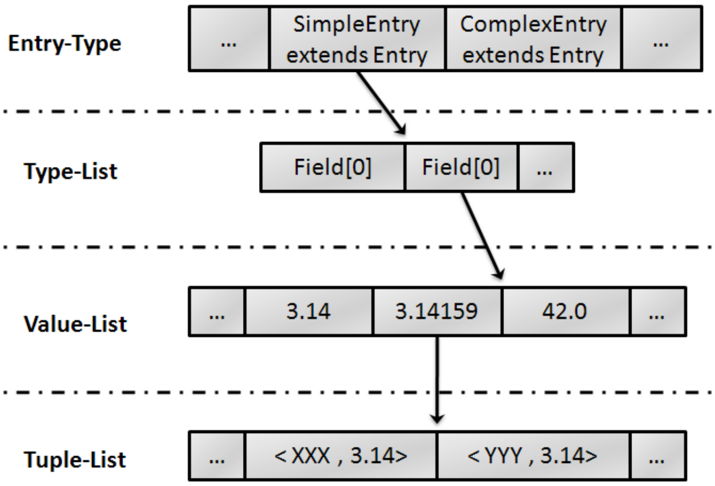


Figure 7.4: JSLindaCoordinator's data structure

7.2.2 Adapted Revised-LindaSelector

The revised LindaSelector, introduced in section 6.3, has been extended by additional features to provide better support for the implementation of JavaSpaces. These features are reflected by the newly added constructors and methods that are listed in the class diagram in figure 6.7.

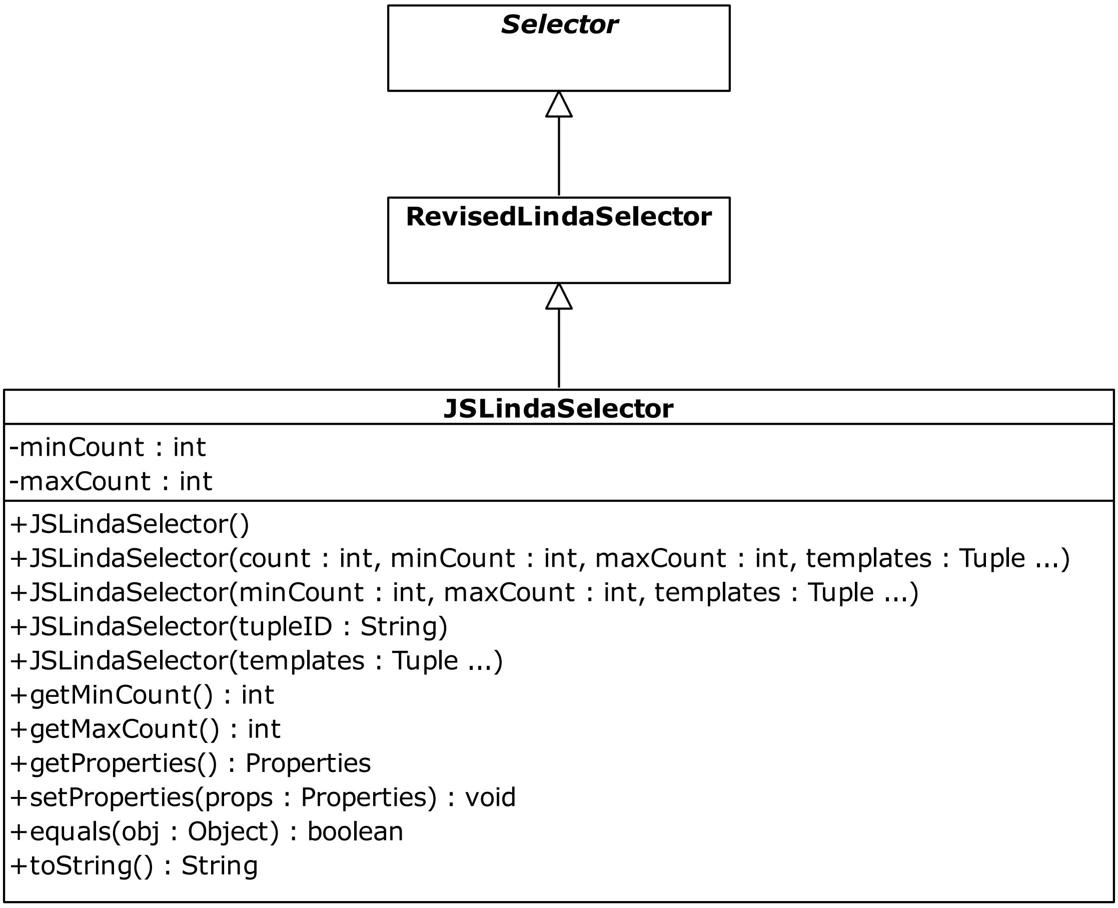

Selector
RevisedLindaSelector
JSLindaSelector
-minCount : int
-maxCount : int
+JSLindaSelector()
+JSLindaSelector(count : int, minCount : int, maxCount : int, templates : Tuple ...)
+JSLindaSelector(minCount : int, maxCount : int, templates : Tuple ...)
+JSLindaSelector(tupleID : String)
+JSLindaSelector(templates : Tuple ...)
+getMinCount() : int
+getMaxCount() : int
+getProperties() : Properties
+setProperties(props : Properties) : void
+equals(obj : Object) : boolean
+toString() : String

Figure 7.5: JSLindaSelector class

The usage of several templates and the definition of a specific tuple to be retrieved is also possible as it was introduced by the revised LindaSelector. The JSLindaSelector extends the revised LindaSelector by additional properties that are required by the JSCoordinator.

Perhaps the most important extensions are the values *minCount* and *maxCount*. These values add, together with the existing value *count*, a new feature to the selector. Based on the relation of these three values the selector has a different behavior relating how many tuples must be found. The possible relations and how the LindaCoordinator behaves in each case are listed below:

- minCount < count < maxCount
  In this case minCount and maxCount are ignored, the LindaCoordinator either returns *count* matching tuples or throws a CountNotMetException.

- minCount ≥ count or
  maxCount ≤ count
  The limiting values *minCount* and *maxCount* are used as lower- and upper-bound when reading or taking multiple tuples. The LindaCoordinator must find minCount tuples, else it would either block or throw a CountNotMetException, but it must not find more than maxCount.

7.3 From Jini Entry to MozartSpaces JSTuple and backward

When converting Jini entries to MozartSpaces tuples attention must be paid to extract the public fields of a Jini Entry and generating an instance of JSTuple out of it. On the one hand the Jini Entry's type is required for instantiate a new JSTuple as well as all public fields of the Jini Entry must be extracted. It is possible to specify the fields to be stored by the JSTuple as well as their ordering, by adding the method *__getFields* to the implementing class of the Jini Entry.

The composition of the tuples that are stored by the JAXS in the JSLindaCoordinator is depicted in figure 7.6. The first field is reserved for a tuple that may contain additional information, like the lease assigned by the JavaSpaces service for the written entry. The tuple is intended to allow interoperability between several JAXS services and is set be ignored by the JSLindaCoordinator. The number of the tuple's remaining fields depends on the size of the entry's public fields that are to be stored. All fields of an entry are stored in marshalled form as specified by the Jini Entry specification.

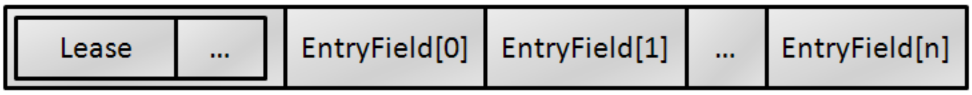


Figure 7.6: Composition of a JSTuple as used by the JAXS

A *write* operation must convert an entry to a tuple before it can be written by the MozartSpaces' write operation to the space. The conversion is done in the following 3 steps:

1. Ascertainment of the entry's class type and instantiation of a new JSTuple with the extracted entry's class type as argument.
2. Ascertainment and extraction of the entry's public fields
3. Each extracted field is stored as a `MarshalledObject`[1] in a separate field of the tuple in the same order as it was extracted in step 2.

---

[1] A *MarshalledObject* is a Java RMI class which provides a constructor that automatically serializes the as parameter provided object and deserializes it again on invocation of its *get* method.

A *read* or *take* operation must reconvert the tuple, returned by the corresponding MozartSpaces operation, to an entry before it can be returned to the invoking client's procedure. The reconversion is the inverse of the previously described conversion and is done in the following 3 steps:

1. Extraction of the stored entry's type.
2. Instantiation of a new empty entry whose type equals the class type of the previously extracted entry.
3. Assignment of the stored values, which are contained by the fields after the second field, to the in step 2 generated entry's fields.

7.4 Implementation of JavaSpace and JavaSpace05 Operations

As introduced in section 4.6, the JavaSpaces API is the interface that must be implemented by services, which want to provide the functionalities of a standard JavaSpaces service. Of course a service could also implement the JavaSpaces05 API, introduced in section 4.7, which extends the functionalities of the original JavaSpaces API by adding support for bulk operations. Although the primary target of this thesis was the implementation of the JavaSpaces API standard, attention was paid from the beginning to allow later the implementation of the JavaSpaces05 API without the need to completely rewrite the sources. Not only that the sources have been prepared for a later implementation of the JavaSpaces05 API, the API has been implemented as part of this thesis too.

As depicted in section 7.1 the JAXS consists of two components that are relevant for the Jini community: the JAXS service and the JAXS service proxy. The service proxy is the front-end through which the Jini clients and services can interact with the JAXS service. Therefore it must implement the JavaSpace or JavaSpace05 interfaces in order to be accessible by them as a JavaSpaces service. The service proxy possesses a reference to the exported remote object of the JAXS service through that it is able to interact with the JAXS service. The diagram in figure 7.7 depicts the symbiosis between the JAXS service, represented by the `JAXSService` class in the implementation, and the service proxy, represented by the `JAXSServiceProxy` class in the implementation. It shows the mapping of the various JavaSpace and JavaSpace05 operations in the service proxy implementation to the corresponding operations in the JAXS service implementation. The `JAXSService` implements the `IRemoteJAXSService` interface, which defines the methods that can be remotely used and accessed by the JAXSServiceProxy. The implementation details of these remote methods will be discussed next.

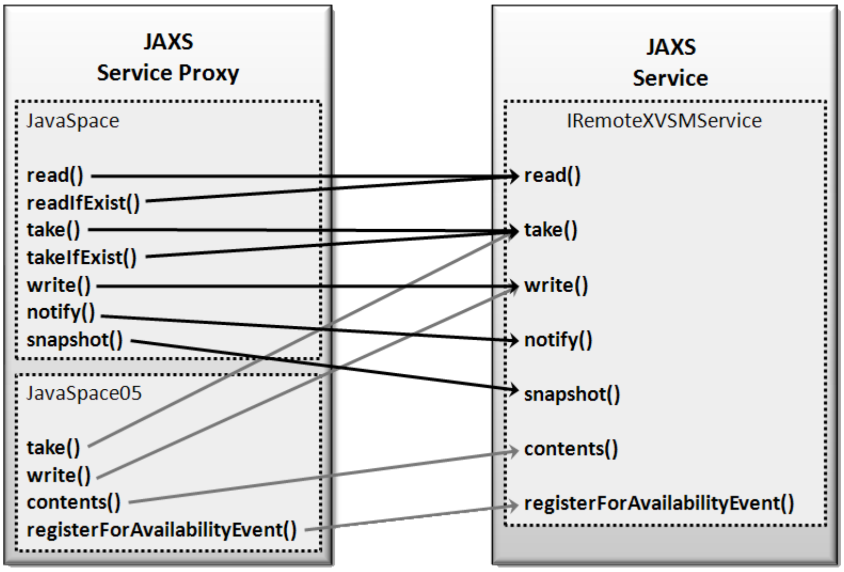


Figure 7.7: Symbiosis between JAXSServiceProxy and JAXSService

7.4.1 Remote JAXSService Operations

As aforementioned, the JAXSService implements the IRemoteJAXSService interface, which defines the operations that can be remotely used by the JAXSServiceProxy to interact with the service. The methods defined by the IRemoteJAXSService interface are listed below in listing 7.8.

All these methods have in common that the passed transaction reference, which is a Jini transaction, has to be converted into a MozartSpaces transaction before it can be further processed. This is done by looking up the transaction at the transaction manager. It will either return the associated MozartSpaces transaction or create a new one if there was no associated so far. More details on the transaction manager's functionality will be discussed in section 7.5. The implementation of the methods itself will be explained next.

write The *write* method takes the passed list of entries, which were converted to tuples before by the service proxy, and writes them into the space by calling the *ICapi.write* method. But before they can be written, a lease must be created for each single entry. The creation of leases is left to the `TupleLeaseManager` (see section 7.7.1 for details). After all leases were created successfully, the entries are written to the space and the references to the created leases returned. In case of failure, any created lease is dropped and a `RemoteException` is thrown.

The JAXSServiceProxy uses the *IRemoteJAXSService.write* to delegate the invocations of its methods *JavaSpace.write* and *JavaSpace05.write* to the JAXSService. In the case of the implemented *JavaSpace.write* method, the *JAXSService.write* method is invoked

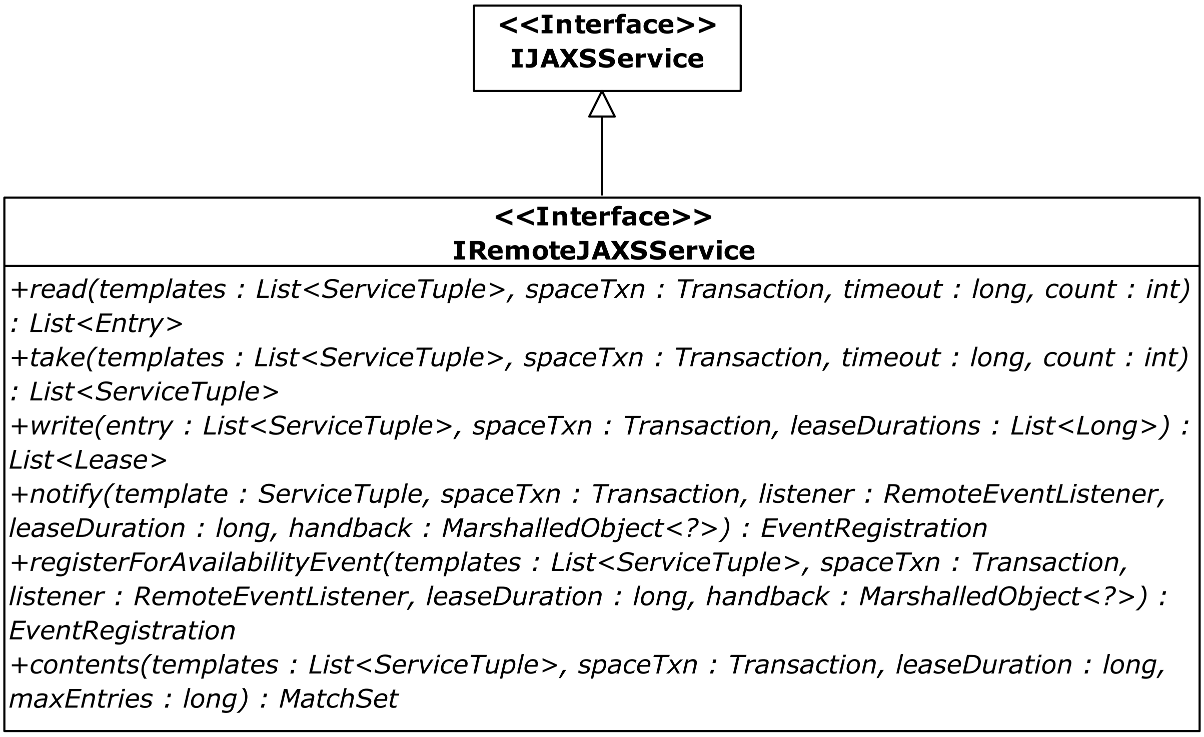

<<Interface>>
IJAXSService
<<Interface>>
IRemoteJAXSService
+read(templates : List<ServiceTuple>, spaceTxn : Transaction, timeout : long, count : int) : List<Entry>
+take(templates : List<ServiceTuple>, spaceTxn : Transaction, timeout : long, count : int) : List<ServiceTuple>
+write(entry : List<ServiceTuple>, spaceTxn : Transaction, leaseDurations : List<Long>) : List<Lease>
+notify(template : ServiceTuple, spaceTxn : Transaction, listener : RemoteEventListener, leaseDuration : long, handback : MarshalledObject<?>) : EventRegistration
+registerForAvailabilityEvent(templates : List<ServiceTuple>, spaceTxn : Transaction, listener : RemoteEventListener, leaseDuration : long, handback : MarshalledObject<?>) : EventRegistration
+contents(templates : List<ServiceTuple>, spaceTxn : Transaction, leaseDuration : long, maxEntries : long) : MatchSet

Figure 7.8: IRemoteJAXSService interface

with a list, which only contains single entry, as argument. In almost the same manner the implemented *JavaSpace05.write* method invokes the *JAXSService.write* method with a list of entries as argument.

read The *read* method simply calls the *ICapi.read* method to find entries in the space that match one or more of the passed templates. This method does not require any additional processing before or after the invocation of the MozartSpaces method's call, except the previously mentioned transaction's conversion.

The *read* method has two arguments, which are directly passed on to the *ICapi.read* method, that are important for the `JAXSServiceProxy`:

- **timeout:** The `JAXSServiceProxy` uses the timeout argument to implement the *JavaSpace.readIfExist* by passing 0 as value. This tells MozartSpaces that the operation shall not block and return anyway.
- **count:** The `JAXSServiceProxy` uses the count argument to implement the *JavaSpace.read* method by passing 1 as value, because the JavaSpace method specifies the return of a single matching argument.

contents The *contents* method is an extension of the read method. It uses the read method to find matching entries, but it does not return the received list immediately. Instead it requests the `MatchSetManager` (see section 7.4.2 for details) to create a new instance of the `ServiceMatchSet` class, containing the received list of entries. The obtained reference of the stored `MatchSet` is then returned to the calling process. This must be done because the *contents* method is the corresponding implementation of the *JavaSpace05.contents* method (see section 4.7.2 for details), which defines the return of a reference to a `MatchSet` that is managed by the implementing service.

take Like the *read* method, the *take* method simply calls the *ICapi.read* method to retrieve entries that match one or more of the passed templates from the space. The arguments, their meaning and the way they are used are the same as for the *read* method. The received entries need to be post processed before they can be returned to the invoking process. This includes the removal of each associated lease from the TupleLeaseManager, because these are no longer required.

notify The *notify* method is used by the JAXSServiceProxy to pass on registration requests that were placed by the corresponding *JavaSpace.notify* method. The registration request is further passed on to the NotificationManager (see section 7.6 for details), which returns a Lease of the registered notification as a voucher that the notification was registered successfully.

registerForAvailabilityEvent Basically the *registerForAvailabilityEvent* works the same way as the *notify* method, except that it allows to provide more than one template at notification registration.

The *JavaSpace.snapshot* method is handled by the `JAXSServiceProxy` itself and is not forwarded to the `JAXSService`. The generated snapshots of entries are simply stored in a HashMap, where they are mapped to their originating entry.

7.4.2 MatchSetManager

The `MatchSetManager` class is responsible to create, store and administrate MatchSets as long as they are valid. It possesses a reference to the MatchSetLeaseManager, which is concerned with the administration of the MatchSets' leases. The MatchSetManager's class diagram is depicted in figure 7.9.

Matchsets are created by the *createMatchSet* method, which requires the caller to provide a list of tuples and the duration that the MatchSet should be leased as arguments on invocation. According to the JavaSpace05 specification a MatchSet (see section 4.7.1 for details) is managed by the service and accessed remotely by the client via the `MatchSet` interface. For this reason a MatchSet is represented within the MatchSet manager by the `ServiceMatchSet` class (depicted in figure 7.10) that contains beside the list of found tuples a unique ID and the MatchSet's lease.

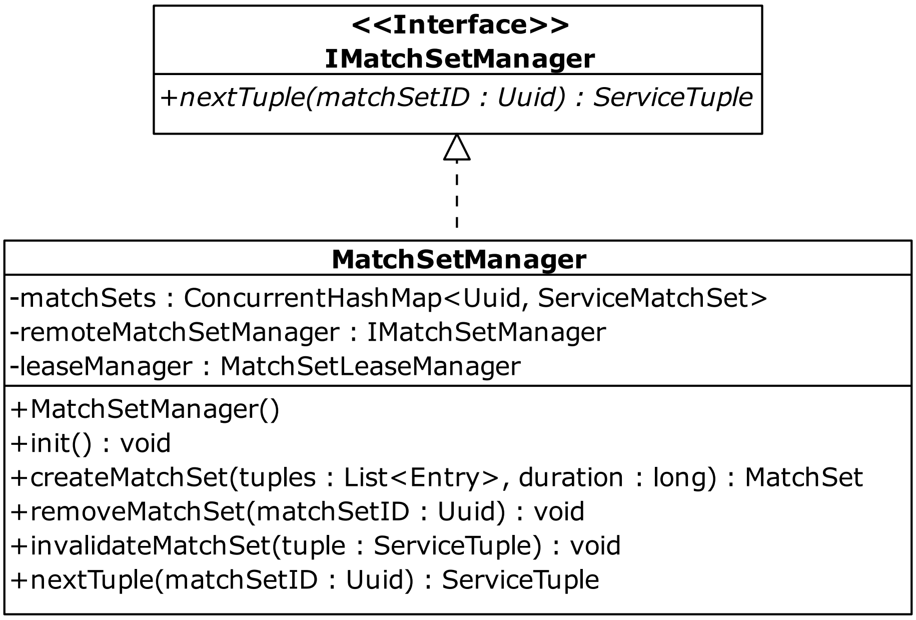

<<Interface>>
IMatchSetManager
+nextTuple(matchSetID : Uuid) : ServiceTuple
MatchSetManager
-matchSets : ConcurrentHashMap<Uuid, ServiceMatchSet>
-remoteMatchSetManager : IMatchSetManager
-leaseManager : MatchSetLeaseManager
+MatchSetManager()
+init() : void
+createMatchSet(tuples : List<Entry>, duration : long) : MatchSet
+removeMatchSet(matchSetID : Uuid) : void
+invalidateMatchSet(tuple : ServiceTuple) : void
+nextTuple(matchSetID : Uuid) : ServiceTuple

Figure 7.9: MatchSetManager class

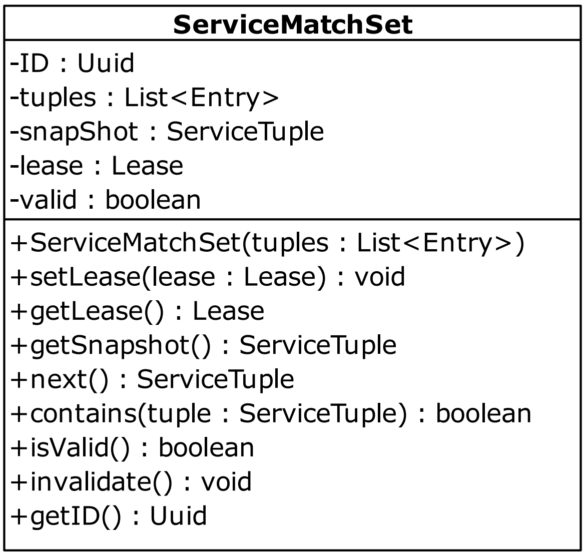


Figure 7.10: ServiceMatchSet class

However the client receives an instance of the `ServiceMatchSetProxy` class, which implements the `MatchSet` interface and contains a reference to the MatchSet manager's remote object. With its instance of the ServiceMatchSetProxy class the client is able to iterate over the content of the MatchSet as long as the MatchSet is not invalidated. A MatchSet is either invalidated if at most one of its contained tuples was removed from the space or after the MatchSet's lease expired.

7.5 From Jini Transactions to MozartSpaces Transactions

Jini transactions are incompatible with MozartSpaces transactions, because Jini transactions are built on a two-phase commit protocol and are centrally managed by a transaction manager. For this reason the JAXS service must map Jini transactions to MozartSpaces transactions. But it is not sufficient to create an association between both transaction types. The Jini transaction specification (see section 3.6 for details) requires that any service that wants to provide support for transactions must additionally join a transaction as a transaction participant and also implement the two-phase commit protocol.

The JAXS service does not implement a Jini transaction manager itself. It only joins a Jini transaction as a transaction participant. For this purpose, Jini transaction manager services like the reference implementation "Mahalo" [53] can be used to create and manage Jini transactions.

In the JAXS service, the mapping and administration of the transactions is left to the `TransactionManager` class. It takes care of the assignment of Jini transactions to MozartSpaces transactions and the creation of MozartSpaces transactions if there was none assigned to a Jini transaction so far. The TransactionManager also joins each used Jini transaction as a transaction participant in order to notice when the transaction is committed or aborted. As a transaction participant it must implement the TransactionParticipant interface and the therein specified methods *prepare*, *commit* and *prepareAndCommit*. The prepare method must return the state of readiness of the transaction participant. In order to do that the TransactionManager must perform a read operation with the MozartSpaces transaction, which is associated to the corresponding Jini transaction, to check whether the transaction is still valid or not. The TransactionManager therefore possesses its own separate container "TXN-Validation" that contains a single entry. If the transaction is still valid the read operation will return the entry and the TransactionManager can send back that it is ready to commit the transaction to the TransactionManager service. If the transaction was no longer valid, the TransactionManager receives an exception and sends back that it must abort the transaction.

The operations workflow of the TransactionManager joining a Jini transaction is depicted in the sequence diagram in figure 7.11 and described below. The `JAXSServiceProxy` was omitted here because it does not play any role in the process of transactions.

A transaction starts with the creation of a new Jini transaction at the Mahalo service by a client and the return of a new transaction jTxn to the client. The client uses the received transaction jTxn to write an entry into the JAXS service. Thereupon, the JAXS service looks up the transaction jTxn at the TransactionManager, which first tries to find an already existing mapping to the transaction jTxn. Because the transaction jTxn is used the first time on the JAXS service the TransactionManager is unable to find an associated MozartSpaces transaction and hence requests MozartSpaces to create a new transaction. The received new MozartSpaces transaction msTxn is mapped to the transaction jTxn in its HashMap. Before it is returned to the JAXS service, the TransactionManager registers itself as a transaction participant of the transaction jTxn at the Mahalo service. The JAXS service creates a lease under the transaction msTxn for the entry at the TupleLeaseManager (see section 7.7.1 for details), before it is written under the transaction msTxn to the space. This is done in order to remove the lease in case of a transaction's rollback and to be able to remove the entry from the space as soon as the lease expired, but before the transaction was already committed. Next the JAXS performs the write operation under the transaction msTxn on the MozartSpaces container "space", which is carried out without any failures and the JAXS service returns to the client.

The operations workflow of the TransactionManager committing a Jini transaction, is depicted in the sequence diagram in figure 7.12 and described next. Again, the `JAXSServiceProxy` was omitted here because it does not play any role in the process of transactions.
After the client has performed all its operations under the transaction jTxn, it requests to commit the transaction jTxn at the Mahalo service, which then asks all the transaction participants that partake in the transaction jTxn, including the TransactionManager, to prepare the transaction for commit. The TransactionManager looks up the transaction jTxn in its HashMap to find the associated MozartSpaces transaction msTxn. With the found transaction it performs a read operation on its container "TXN-Validation" to check whether the transaction is still valid or not. In this case the read operation returns successfully and the TransactionManager sends back that it is ready to commit the transaction. In the last step the Mahalo service requests the transaction participants of the transaction jTxn, including the TransactionManager, to commit the transaction jTxn. After the TransactionManager looked up the mapped transaction msTxn for the transaction jTxn, it requests MozartSpaces and after that the TupleLeaseManager to commit the transaction msTxn. The TupleLeaseManager must remove the transaction reference from the lease object. Otherwise it would receive an exception as soon as it tries to remove the entry from the space. Finally the TransactionManager reports back that the transaction was committed successfully.

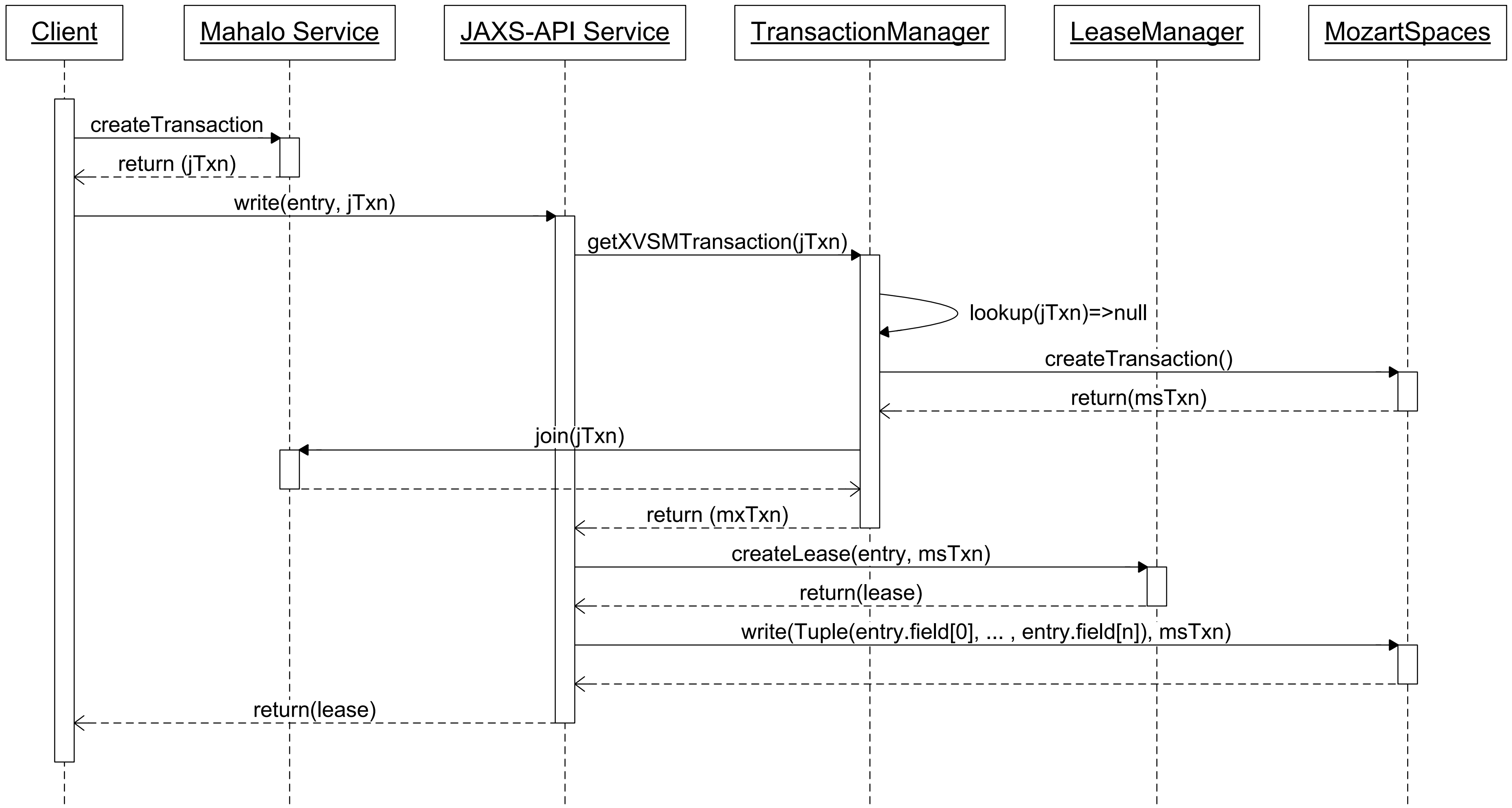


Figure 7.11: Workflow of the TransactionManager joining a transaction

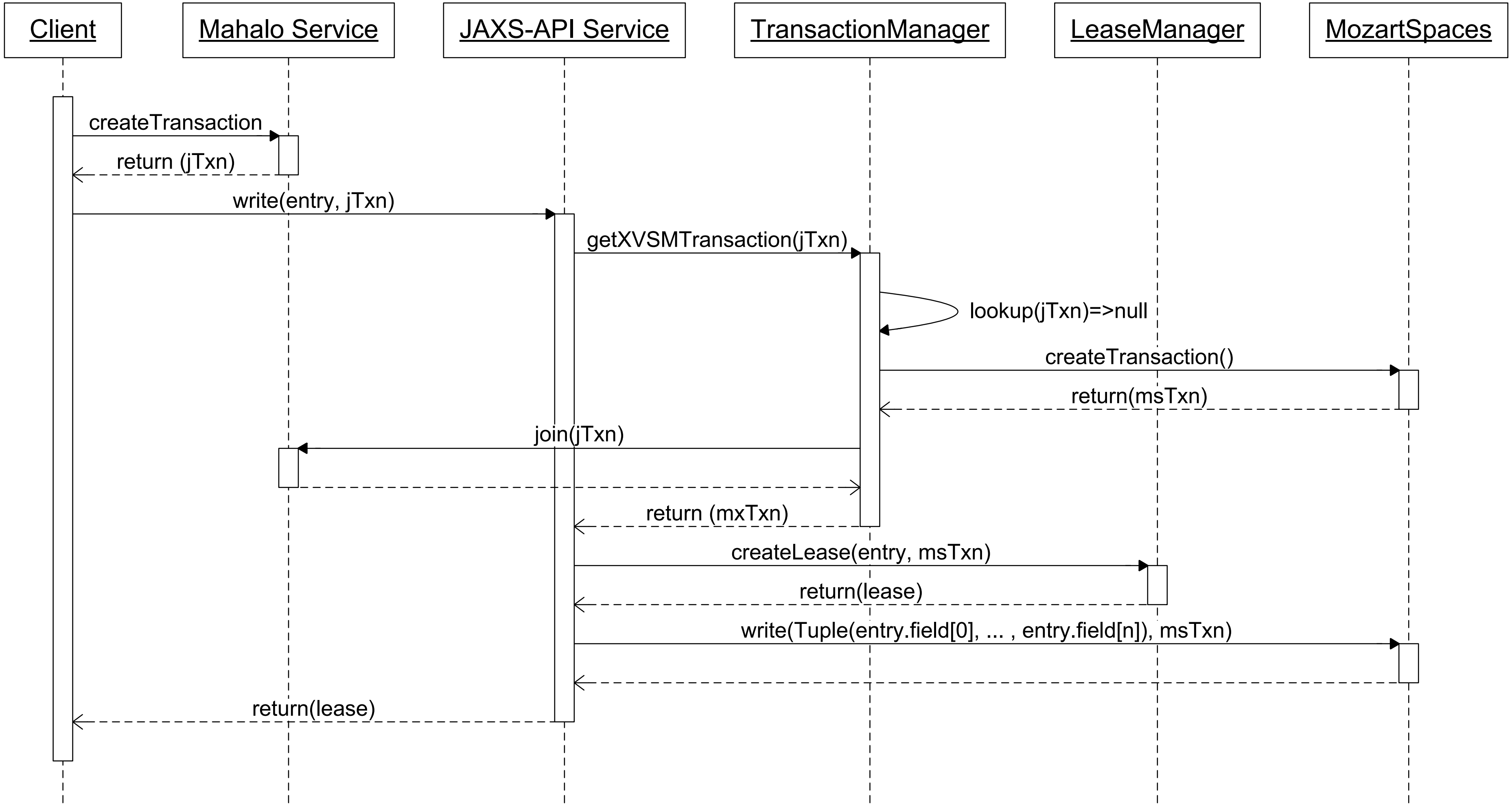


Figure 7.12: Workflow of the TransactionManager committing a transaction

If the transaction is rolled back for any reason the TransactionManager issues MozartSpaces to rollback the transaction and requests the lease manager to drop the leases associated with the transaction by calling the *LeaseManager.rollback* method.

7.6 Notifications

The notification feature provided by MozartSpaces is not applicable to the notification specified by the JavaSpaces API specification, which specifies that clients must provide a template when registering their notification. The template is required to inform the client only when entries, matching the provided template, are written. Template matching works the same way as it does in a common read or take operation. For this reason a custom notification module for the JAXS service was implemented that complies to the JavaSpaces specifications and consists of as many existing MozartSpaces' features as possible. The architecture of the implemented notification module is shown in figure 7.13.

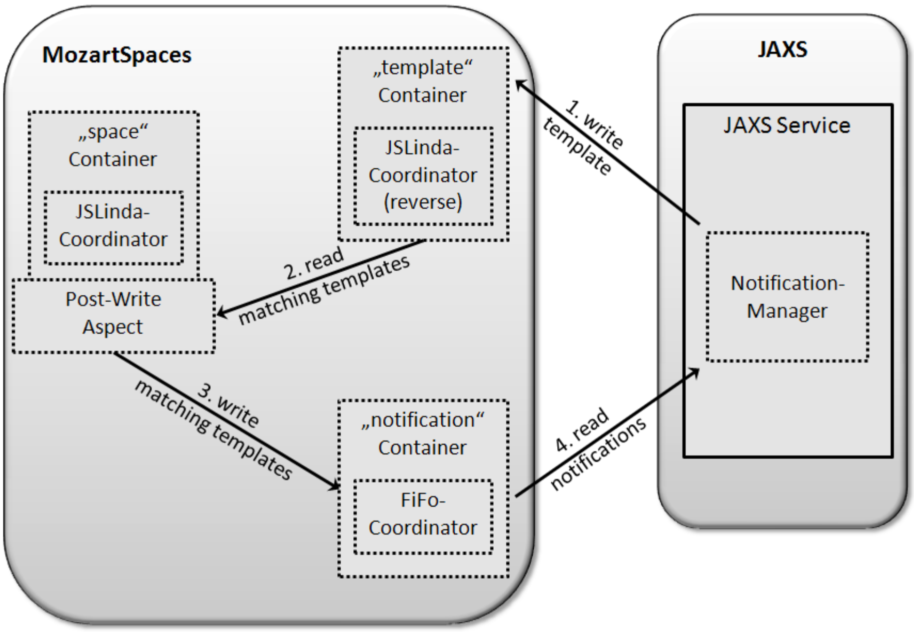


Figure 7.13: Notification architecture

Template Container The "template" container uses a JSLindaCoordinator with the "reverseMatching" option turned on to manage its content. The reverse matching is required because the coordinator has to work the other way round as it would normally do. This way, templates are stored like entries and entries are used like templates by the coordinator.

Notifications Container The “notification” container uses a FiFoCoordinator to manage its content. It contains tuples that are composed of the MozartSpaces transaction, which was used by the write operation, and all templates that are matching the just written entry.

Notification Aspect The `NotificationAspect` is an extension of the LocalAspect class that implements the *postWrite* method. On invocation the *postWrite* method performs a *read* operation with the just written entry, as the template to be matched, on the template container. The template container returns the matching templates, which are placed together with the transaction that was used by the write operation, into a newly created tuple (the tuple’s composition is depicted in figure 7.14). The resulting tuple then is stored in the notification container.

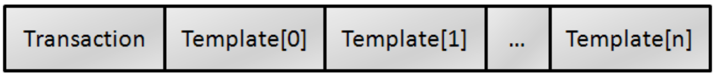


Figure 7.14: Notification tuple composition

Notification Participant Each registration of a client is represented by an instance of the `NotificationParticipant` class. It stores all the information that was provided by the client on its registration (the transaction, a reference to the client’s implementation of the `RemoteEventListener` interface and the handback object) including the event the client has registered for.

Notification Manager The notification manager administrates the registered notification and uploads each new template into the template container. A separate thread is engaged to carry out a blocking *take* operation on the notification container in an infinite loop, in order to receive the matching templates from the notification aspect. As soon as the blocking take returns any result, a separate thread is charged with the notification of the clients, which are registered with the same template as it was returned, about the occurred event.

One problem to be solved was the different behavior of the *JavaSpace.notify* and the *JavaSpace05.registerForAvailabilityEvent* method. The difference between them is that the JavaSpace method allows the registration of a single template only, whereas the JavaSpace05 method allows the registration of multiple templates. Each registration requires the assignment of an event ID with respect to all other existing active registrations and the way they were registered. This means that events registered via the *JavaSpace.notify* method must be handled separately from those registered via the *JavaSpace05.registerForAvailabilityEvent* method.
Registered notifications are distinguished according to the way they were registered and represented by two classes, both extending the `AbstractEvent` class:

- The `Event` class, representing notifications registered via the *JavaSpace.notify* method.
- The `EventSet` class, representing notifications registered via the *JavaSpace05.registerForAvailabilityEvent* method

Within the notification manager the correlation between events and the notification participants is administrated by the `EventManager` class. For this purpose the `EventManager` possesses a HashMap where a list of notification participants is mapped to their associated event. Events exist as long as there is at least one notification participant associated with it, whereas the notification participants are removed as soon as their lease expired. Hence it might happen that an event is never dropped, if the presence time of coming and going notification participants overlaps. Once an event is dropped a new registration for the same kind will result in the creation of a new event, with a different ID as the one before.

7.7 Leasing

The JavaSpaces API specification specifies that entries, MatchSets and notifications must be leased by the implementing service. This should prevent that unneeded resources are unnecessarily wasting important system resources. MozartSpaces does not yet feature the lease of resources, hence the entire lease management had to be implemented within the JAXS service. Although it may seem naturally to implement the lease's management with the aid of aspects within MozartSpaces, it was not realized this way because it would have violated the MozartSpaces specification. On the one hand MozartSpaces does not admit the use of custom threads within aspects and on the other hand it does not admit to directly access aspects from the outside of MozartSpaces too. If an aspect is assigned with the leases' management, it must check the leases in certain intervals. This would require the use of threads, which would violate the specification of MozartSpaces. According to the Jini lease specification, the returned Lease object contains a reference to the remote lease manager, enabling the client to manage its leases. Allowing the client to access the aspect, which is assigned with the leases' management, would again violate the specification of MozartSpaces.

The leases' management for the different resources is nearly the same. Nevertheless they are managed seperately from each other by its own lease manager. Therefore the JAXS owns three different lease managers, all extending the `AbstractLeaseManager` class. The `AbstractLeaseManager` class takes care of the main work, the actual management of the leases. It creates new leases for resources and stores them in its own HashMap associated to the leased resource's ID. A leased resource is represented internally by the `AbstractLeasedResource` class, which is implemented seperately for each type of leased resource. It contains only the ID of the resource that is leased and the time when the lease will expire.

The `AbstractLeaseManager` class possesses its own thread that is concerned with checking whether a lease resource has expired and its associated resources must be dropped. If an expired lease is found the *handleTimout* method is called, which has to be implemented by all classes that are implementing the `AbstractLeaseManager` class, in order to remove the expired lease.

The three different lease managers TupleLeaseManager, NotificationLeaseManager and MatchSetLeaseManager that are used by the JAXS will be discussed next, followed by a short description about the leasing of the JAXS service at a lookup service.

7.7.1 Lease Managers

TupleLeaseManager The `TupleLeaseManager` class is responsible for managing the leases of the tuples stored by MozartSpaces. In contrast to the other lease managers the TupleLeaseManager must access MozartSpaces in order to be able to remove a tuple whose lease has expired. The leased tuples are represented by the `TupleLeasedResource` class, an extension of the `AbstractLeasedResource` class, which contains the ID of the stored tuple and the transaction that was used to write the tuple into the space. The transaction is removed again as soon as the transaction was committed successfully.

Attention had to be paid for the case that a tuple was removed from the space again before the lease expired, because the lease was no longer valid. For this reason the JAXS service removes the lease for each taken tuple from the TupleLeaseManager.

In the case that a *write* or *take* operation is performed under a transaction, the TupleLeaseManager owns its own `LeaseHistory` class, which extends the `AbstractHistory` class, to be able to rollback the adding or removal of a lease in case of that the transaction is aborted.

NotificationLeaseManager The `NotificationLeaseManager` class is responsible for managing the leases of the registered notifications. It owns a reference to the NotificationManager to be able to remove the expired notification. The `NotificationLeasedResource` class, an extension of the `AbstractLeasedResource` class, represents the leased notification within the NotificationLeaseManager.

MatchSetLeaseManager The `MatchSetLeaseManager` class is responsible for managing the leases of the created MatchSets. It owns a reference to the MatchSetManager to be able to remove the expired MatchSet. The `MatchSetLeasedResource` class, an extension of the `AbstractLeasedResource` class, represents the leased notification within the MatchSetLeaseManager.

The class diagram in figure 7.15 gives a better overview on the class hierarchy of the various lease managers.

7.7.2 JAXS Service Leasing

The JAXS as a Jini service owns a lease for its registration at a lookup service. The lease is updated periodically to ensure that the registration is not dropped by the lookup service as long as the service is running. For this purpose Jini already provides the `LeaseRenewalManager` class, a tool that periodically updates leases.

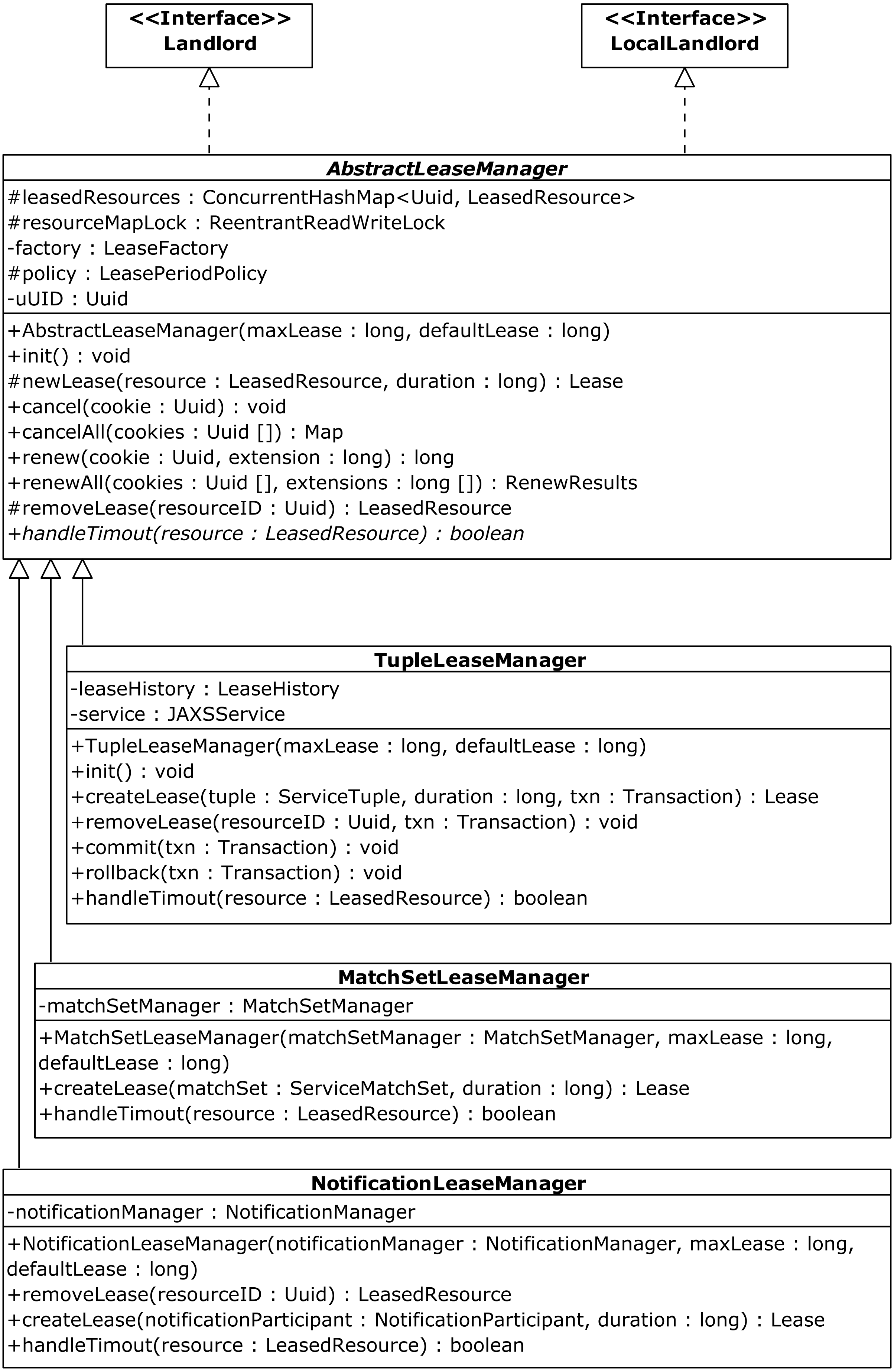

<<Interface>>
Landlord
<<Interface>>
LocalLandlord
AbstractLeaseManager
#leasedResources : ConcurrentHashMap<Uuid, LeasedResource>
#resourceMapLock : ReentrantReadWriteLock
-factory : LeaseFactory
#policy : LeasePeriodPolicy
-uUID : Uuid
+AbstractLeaseManager(maxLease : long, defaultLease : long)
+init() : void
#newLease(resource : LeasedResource, duration : long) : Lease
+cancel(cookie : Uuid) : void
+cancelAll(cookies : Uuid []) : Map
+renew(cookie : Uuid, extension : long) : long
+renewAll(cookies : Uuid [], extensions : long []) : RenewResults
#removeLease(resourceID : Uuid) : LeasedResource
+handleTimout(resource : LeasedResource) : boolean
TupleLeaseManager
-leaseHistory : LeaseHistory
-service : JAXSService
+TupleLeaseManager(maxLease : long, defaultLease : long)
+init() : void
+createLease(tuple : ServiceTuple, duration : long, txn : Transaction) : Lease
+removeLease(resourceID : Uuid, txn : Transaction) : void
+commit(txn : Transaction) : void
+rollback(txn : Transaction) : void
+handleTimout(resource : LeasedResource) : boolean
MatchSetLeaseManager
-matchSetManager : MatchSetManager
+MatchSetLeaseManager(matchSetManager : MatchSetManager, maxLease : long, defaultLease : long)
+createLease(matchSet : ServiceMatchSet, duration : long) : Lease
+handleTimout(resource : LeasedResource) : boolean
NotificationLeaseManager
-notificationManager : NotificationManager
+NotificationLeaseManager(notificationManager : NotificationManager, maxLease : long, defaultLease : long)
+removeLease(resourceID : Uuid) : LeasedResource
+createLease(notificationParticipant : NotificationParticipant, duration : long) : Lease
+handleTimout(resource : LeasedResource) : boolean

Figure 7.15: Lease managers' class diagram

Chapter 8

Evaluation and Benchmarking

This section evaluates first in which way the implemented JavaSpaces API for XVSM-Space (JAXS), as part of this thesis, complies with the JavaSpaces API standard specification. After that a comparison between JAXS and the JavaSpaces implementations GigaSpaces, Blitz and Outrigger is given. In the end, a description of possible enhancements of JAXS and the concept of a further feature are presented.

8.1 JavaSpaces API Compliance

The implemented JavaSpaces-XVSM API (JAXS) as part of this thesis complies in almost every point to the JavaSpaces specification. The compliance was approved by testing the API with own test cases (see appendix B) and by running 3rd party applications (e.g. the applications listed in [42]) that use the JavaSpaces API standard.
JAXS does not comply with the JavaSpaces specification in the following points:

- The JavaSpace interface specification prescribes for the *readIfExist* and *takeIfExist* methods that they should block if a matching entry was found. The problem is that its transactional state does not terminates so far until either the transaction terminates, or the given timeout has expired (see section 4.6 for details). The execution of test cases J.11 and J.13 (see section B.2) showed that JAXS does not comply with the specification in this particular cases.
  Explanation: JAXS uses MozartSpaces as storage for entries and as a transaction manager. MozartSpaces is accessed by JAXS via the ICapi interface and utilizes the MozartSpaces' methods *read* and *take*, which do not support the required type of blocking.
- The JavaSpace05 specification prescribes for the *registerForAvailabilityEvent* method the detection of the transition of an entry from invisible to visible and from unavailable to

available (see section 4.7.2 for details). The execution of test case N.10 (see section B.5) showed that JAXS does not comply with the specification in this particular case.

Explanation: JAXS supports the notification of newly added entries to the space but does not support the transitions specified by the JavaSpace05 specification. This is again due to the fact that MozartSpaces is used both as storage for entries and as transaction manager. The internal process flow of MozartSpaces does not allow the implementation of the required feature.

8.2 Benchmarks

Benchmarks were made with the objective to compare the JavaSpaces API implemented on top of MozartSpaces against the JavaSpaces API implementations (see section 2.4.1) GigaSpaces, Blitz and Outrigger. For this purpose, the amount of time was measured that each JavaSpaces implementation required to complete write, read and take operations. These operations were performed separately and independently from each other on all JavaSpaces implementations. Multiple measurements were made, each with a different number of entries.

To be able to insert entries into the space the Jini `Entry` interface has been implemented by the SimpleEntry class, the class diagram is depicted in figure 8.1, which contains three public fields: Integer, String, and a Boolean field. Instances of `SimpleEntry` are used on the one hand to fill the space and on the other hand as template to perform read and take operations. Table 8.1 depicts the different entries and their values that were used for this purpose.

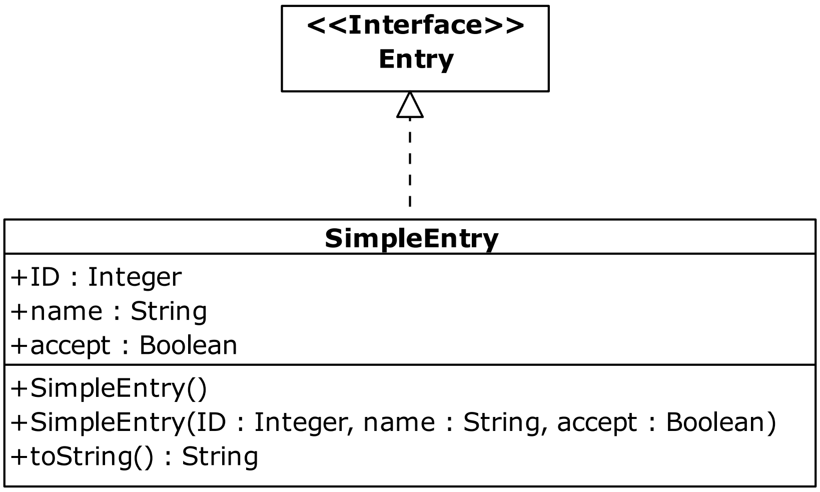


Figure 8.1: SimpleEntry class

The “normal entry” is written (n - 1) times, where n is the number of entries to be written within a measurement of the *write* operation. The “predefined entry” is only written once and before

	Normal entry	Predefined entry	Template entry
Integer	[0 . . . (n - 2)]	42	null
String	“Tuple”	“Benchmark”	null
Boolean	true	true	null

Table 8.1: Values assigned to SimpleEntry instances

the normal tuples by the *write* operation, because this special tuple is at the same time the one for which the read operation will look for during the measurement of the *read* operation. After the *write* operation’s measurement a total number of n tuples are contained in the space. All entries were inserted with the maximum available lease time to avoid the expiration of entries and the necessity to renew the leases, which would have influenced the measurements, during the benchmark.

Now that the space contains n entries, including the predefined entry, the performance of the *read* operation was measured. For this purpose the predefined entry is required to measure the time needed for each space implementation to find the predefined entry out of the previously written n entries.

In the final step the time that each space implementation takes to perform as many *take* operations as entries contained in the space is measured. The “template entry”, depicted in table 8.1, is used to perform the take operations because all fields of the template entry are unassigned (*null*) and hence allow to take all entries that were written previously by the *write* operation’s measurement.

As a general restriction the standard *JavaSpace* interface (see section 4.6) was only used to write, read or take entries. For this reason, the operations are further constrained to work with a single entry respectively. All measurements were repeated 10 times, independently from each other, and the repetitions’ mean value was used to finally generate the diagrams.

The relevant characteristics of the system, on which all benchmarks where performed, are listed in table 8.2.

Processor	AMD Athlon™ 64 X2 Dual Core 6400+ (each core with a frequency of 3.2GHz)
Main memory	2GB
OS	Ubuntu 8.04 32bit
Java™ Version	1.6.0_07-b06

Table 8.2: Computer’s characteristics used for all benchmarks

In the following, the results of the benchmark will be discussed, starting with the results of the *write* performance measurement in section 8.2.1, followed by the results of the *read* performance measurement in section 8.2.2 and then by the results of the *take* performance measurement in section 8.2.3. At last an evaluation is given in section 8.2.4 about the performance of the implemented JAXS compared to the performance of the JavaSpaces implementations GigaSpaces, Blitz and Outrigger.

8.2.1 Write Performance Measurement

The benchmark results of *write* operations performed on the JavaSpaces implementations are presented in figure 8.2. It shows that the performance of the implemented JAXS is nearly the same as the performance of the other JavaSpaces implementations up to 1000 sequentially written tuples. JAXS requires much more time to sequentially write more than 1000 tuples as the other spaces. On the one hand this is due to the overhead required by the JSLindaCoordinator's data structure to index the tuples. On the other hand it is slower because the JavaSpaces entries have to be transformed into MozartSpaces tuples before they can be further processed.
Outrigger and Blitz are getting slower than GigaSpaces above 5000 sequentially written tuples. GigaSpaces showed the best performance in this benchmark.

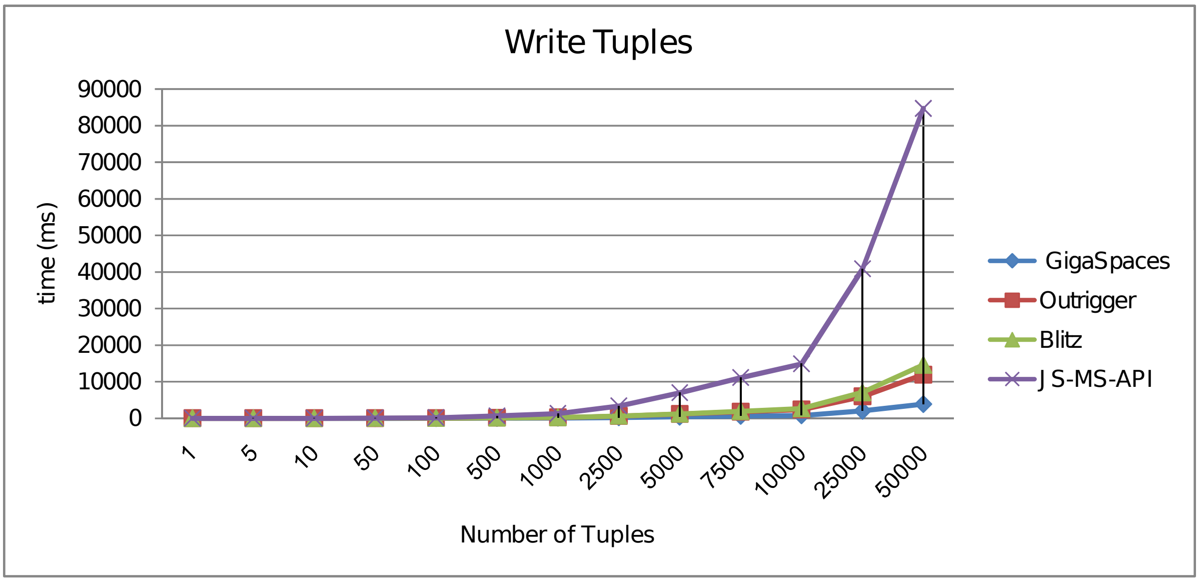


Figure 8.2: Benchmark results of the "write" operation

8.2.2 Read Performance Measurement

The results of the *read* operation's benchmark performed on the JavaSpaces implementations are presented in figure 8.3. Nearly all spaces show a constant performance when reading the predefined entry out of 500 or more entries, GigaSpaces even showed throughout a constant performance. JAXS is slower than the other spaces because of the required time of transforming the template (a JavaSpaces entry instance) into a MozartSpaces tuple and the result back to a JavaSpaces entry than the other spaces

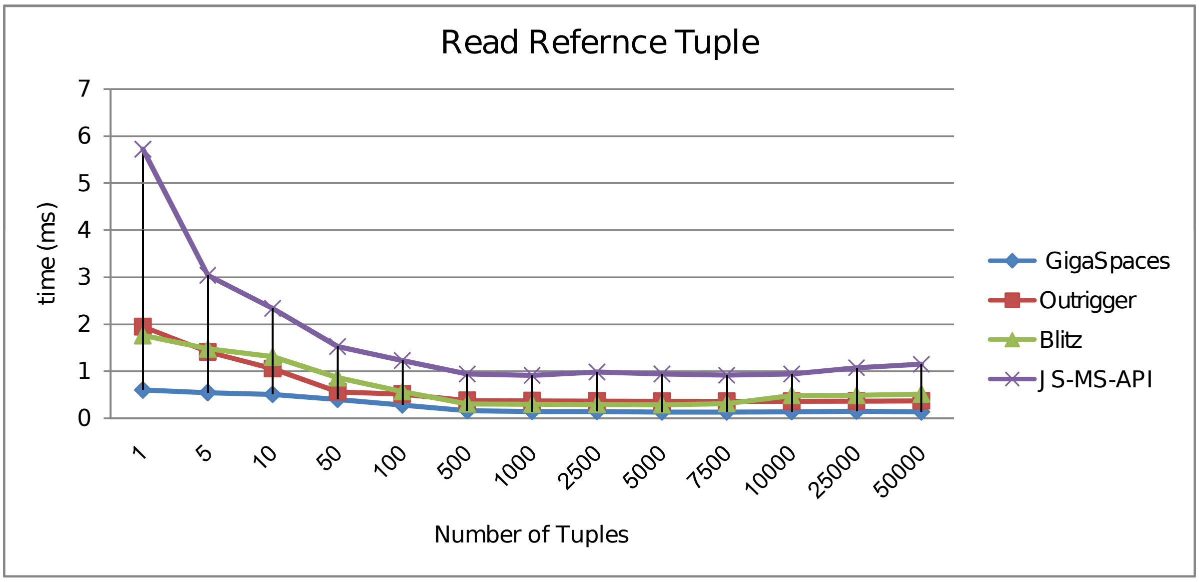


Figure 8.3: Benchmark results of the “read” operation

8.2.3 Take Performance Measurement

The results of the *take* operation’s benchmark performed on the JavaSpaces implementations are presented in figure 8.4. The spaces are almost equally fast until a number of 5000 stored tuples and as from 7500 stored tuples upward the performance is clearly diverging. Blitz shows the worst performance of all, in this benchmark because it requires more than 2 minutes to sequentially take 25000 tuples and even more than 5 minutes for 50000 tuples. JAXS is slowing down continuously with increasing tuple size.

8.2.4 Benchmark Evaluation

Overall JAXS is slower than the other spaces, except Blitz in the “take” operation’s benchmark, but shows a good performance even with more than 10000 tuples as well. The sole bottleneck is the write operation, just as with the revised LindaCoordinator (compare with the revised LindaCoordinator benchmark in section 6.4) that is using nearly the same data structure as the JSLindaCoordinator.

Principally JAXS is however slower than the other spaces, because it acts as a middleman between the JavaSpaces clients and the MozartSpaces and as such it is interacting on the JavaSpaces side via RMI with the XVSMServiceProxy, used by the JavaSpaces clients, and interacting on the other hand with MozartSpaces via TCP/IP.

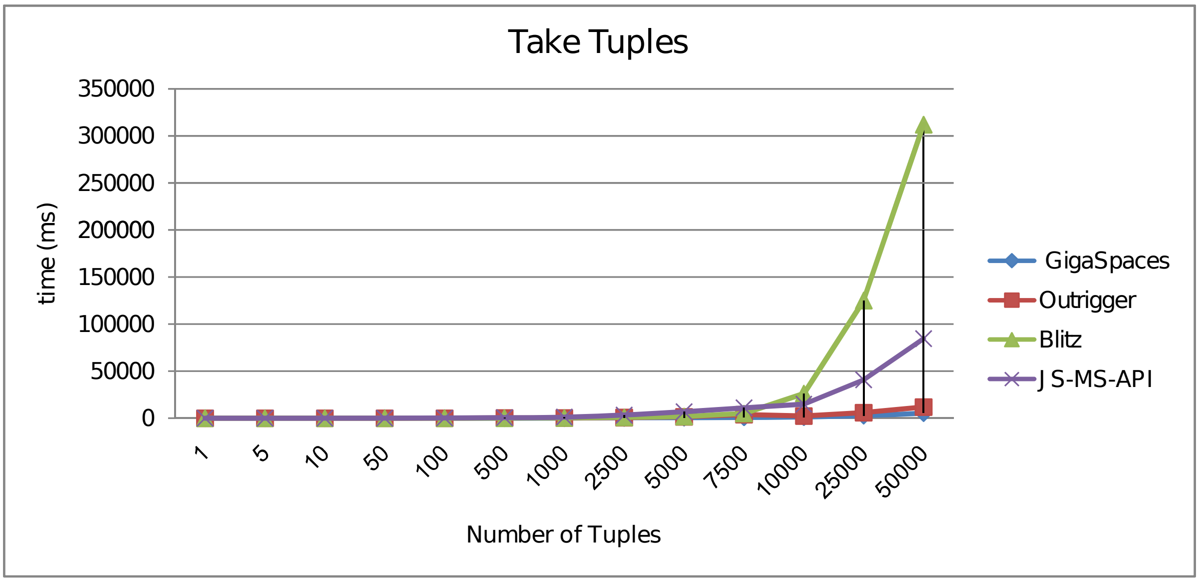


Figure 8.4: Benchmark results of the “take” operation

In overall however, it has to mentioned that MozartSpaces is a premature implementation with a lot potential of improving its implementation. The aspect of benchmarking an improved version of MozartSpaces can be seen as an additional future work resulting in hopefully better measurements than the current one shows.

8.3 Future Work

Performance Enhancement To increase the performance of the implemented JAXS, the administration and management of the Jini leases should be delegated to MozartSpaces. In concrete terms, by adding the feature of leased tuples to MozartSpaces, the management of entries would be centralized. Thus the omission of the “middleman”, would not only simplify the job of JAXS, but also increase its performance.

ActiveCoordinator Based on the concepts of the *eval* operation introduced by the Linda coordination model (see section 2.3), and the P2P characteristics of MozartSpaces the idea came up to add a new feature similar to Linda model’s eval operation. This requires the implementation of a new coordinator “ActiveCoordinator” for MozartSpaces that only accepts classes implementing the `ActiveEntry` interface, which extends the MozartSpaces `Entry` interface and contains the abstract method *evaluate* as depicted in listing 8.5. The *evaluate* method returns a list of instances of classes implementing the `Runnable` interface[1]. The coordinator takes this list and distributes the contained classes, which implement the Runnable interface, across other running MozartSpaces

---

[1] Any class implementing the `Runnable` is executeable by a thread [47].

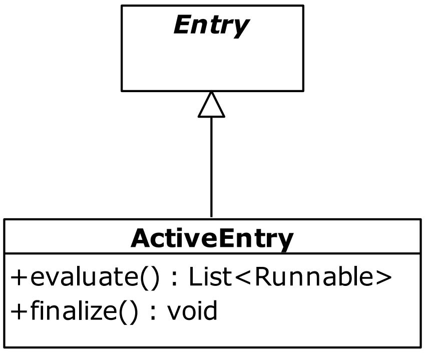


Figure 8.5: ActiveEntry class

that offer free computing time to other participants. Runnable instances can use MozartSpaces just like normal clients are using it. This means that they could also place instances of ActiveEntries into an ActiveCoordinator to further subdivide its computation. The results are returned to the source coordinator which then calls the *finalize* method of the ActiveEntry instance to do final computations, before the ActiveEntry instance can be retrieved again from the coordinator, containing the computation result(s).
The ActiveCoordinator is self-organized because it is responsible to distribute the Runnable instances across other MozartSpaces, offering free computation time, and to collect the results. The distribution could take place on the basis of the available capacity and characteristics of the MozartSpaces.

Great care must be taken to prevent the execution of malicious code by the Runnable instances contained by the ActiveEntry, as well as exception handling of the Runnable instances that must be returned to the source coordinator.

The introduced ActiveCoordinator could be offered by JAXS as an extension to the JavaSpaces API standard. Just like Jini Entries are the analog of MozartSpaces Tuples, an analog to the ActiveEntry interface must exist for the extended JavaSpaces API to be able to forward the client's request to MozartSpaces.

Chapter 9

Conclusion

The design and implementation of the JavaSpaces API standard for XVSM, especially for MozartSpaces, has been presented in this diploma thesis. As a result, already existing JavaSpaces based systems and applications may use MozartSpaces without the necessity to adapt or rewrite their source code. Furthermore the implementation shall demonstrate the flexibility and extensibility of MozartSpaces.
The implementation uses preferably the features of MozartSpaces and the possibility to extend or modify them, respectively. As part of the implementation it was necessary to develop a new coordinator, complying with the Jini Entry specification, particularly with regard to the prescribed template matching. In addition, the support for Jini Transactions, Events and Leases were considered in order to create the outward impression of a complete JavaSpaces implementation.
The resulting JavaSpaces frontend was in the final stages not only tested for its compliance with the JavaSpaces API specification, but also compared in a set of benchmarks against the other JavaSpaces implementation GigaSpaces, Blitz and Outrigger. The benchmarks showed that the presented implementation is not much slower than the others and is even faster in a few particular cases.

Another objective of this thesis was the enhancement of the MozartSpaces' LindaCoordinator in order to increase the coordinators performance. In the scope of the LindaCoordinator's revision a new data structure was developed that allows fast retrieval of tuples, matching one ore more specified templates. Both LindaCoordinators, the original and the revised one, were compared in a set of benchmarks, which proved that read and take operations have been successfully speed up by the sophisticated data structure that exploits the characteristics of the Linda coordination model.

This thesis comprises a detailed description of the Linda coordination model, the basic idea behind space based system, including a description of the few operations that can be used to build powerful distributed applications.

To conclude, MozartSpaces now supports an additional API standard, the JavaSpaces API, beside its other existing API such as JMS-API, JavaScript-API and the Java high level API. Furthermore, MozartSpaces is applicable as a Jini enabled service by Jini clients and other Jini enabled services.

Chapter 10

Bibliography

[1] ActiveSpace. Website. `http://activespace.codehaus.org`; last visited on September 24th 2008.

[2] Almaden Research Center. Website. `http://www.almaden.ibm.com`; last visited on September 26th 2008.

[3] Apache River Project. Website. `http://incubator.apache.org/river/RIVER/index.html`; last visited on July 12th 2008.

[4] Blitz. Website. `http://wiki.community.objectware.no/display/blitz`; last visited on September 20th 2008.

[5] GigaSpaces. Website. `http://www.gigaspaces.com`; last visited on September 20th 2008.

[6] Institute of Computer Languages at the Vienna University of Technology. Website. `http://www.complang.tuwien.ac.at`; last visited on October 10th 2008.

[7] JavaSpaces Service Specification v2.1. Website. `http://java.sun.com/products/jini/2.1/doc/specs/html/js-spec.html`; last visited on July 16th 2008.

[8] JCache. Website. `http://jcp.org/en/jsr/detail?id=107`; last visited on September 24th 2008.

[9] LighTS. Website. `http://lights.sourceforge.net`; last visited on September 23th 2008.

[10] LIME. Website. `http://lime.sourceforge.net`; last visited on September 23th 2008.

[11] MozartSpaces. Website. `http://www.mozartspaces.org`; last visited on September 20th 2008.

[12] muCode. Website. `http://mucode.sourceforge.net`; last visited on September 23th 2008.

[13] OpenSpaces. Website. `http://www.openspaces.org`; last visited on September 20th 2008.

[14] ReSpecT. Website. `http://alice.unibo.it/xwiki/bin/view/ReSpecT`; last visited on September 24th 2008.

[15] Space Based Computing Group. Website. `http://www.complang.tuwien.ac.at/eva/SBC-Group/sbcGroupIndex.html`; last visited on September 24th 2008.

[16] Sun Microsystems. Website. `http://www.sun.com`; last visited on July 1st 2008.

[17] TuCSoN. Website. `http://alice.unibo.it/xwiki/bin/view/TuCSoN`; last visited on September 24th 2008.

[18] XML Query Language. Website. `http://www.w3.org/TandS/QL/QL98/pp/xql.html`; last visited on September 22th 2008.

[19] S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *Computer*, 19(8):26–34, Aug. 1986.

[20] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.

[21] Ken Arnold. The Jini Architecture: Dynamic Services in a Flexible Network. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 157–162, New York, NY, USA, 1999. ACM.

[22] Davide Balzarotti, Paolo Costa, and Gian Pietro Picco. The LighTS Tuple Space Framework and its Customization for Context-Aware Applications. *Web Intelli. and Agent Sys.*, 5(2):215–231, 2007.

[23] Philip Bishop. *Javaspaces in Practice*. Addison-Wesley, Boston, 2003.

[24] Johann Blieberger, Johann Klasek, and eva Kühn. Ada Binding to a Shared Object Layer. In *Ada-Europe '99: Proceedings of the 1999 Ada-Europe International Conference on Reliable Software Technologies*, pages 263–274, London, UK, 1999. Springer-Verlag.

[25] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. MARS: A Programmable Coordination Architecture for Mobile Agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[26] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Mobile Agent Coordination for Distributed Network Management. *J. Netw. Syst. Manage.*, 9(4):435–456, 2001.

[27] Paolo Costa, Luca Mottola, Amy L. Murphy, and Gian Pietro Picco. TeenyLIME: Transiently Shared Tuple Space Middleware for Wireless Sensor Networks. In *MidSens '06: Proceedings of the international workshop on Middleware for sensor networks*, pages 43–48, New York, NY, USA, 2006. ACM.

[28] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. TinyLIME: Bridging Mobile and Sensor Networks through Middleware. In *PERCOM '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, pages 61–72, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Enrico Denti, Antonio Natali, and Andrea Omicini. On the Expressive Power of a Language for Programming Coordination Media. In *1998 ACM Symposium on Applied Computing (SAC'98)*, pages 169–177, Atlanta, GA, USA, 27 February – 1 March 1998. ACM. Special Track on Coordination Models, Languages and Applications.

[30] Russell Dyer. *MySQL in a Nutshell, 2nd Edition*. O'Reilly, 2008.

[31] James Elliott, Tim O'Brien, and Ryan Fowler. *Harnessing Hibernate*. O'Reilly, 2008.

[32] eva Kühn, R. Mordinyi, and C. Schreiber. An Extensible Space-based Coordination Approach for Modeling Complex Patterns in Large Systems. *3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Special Track on Formal Methods for Analysing and Verifying Very Large Systems*, 2008.

[33] eva Kühn. Fault-Tolerance for Communicating Multidatabase Transactions. In *Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS), Wailea, Maui, Hawaii. ACM, IEEE.*, volume 2, pages 323–332, January 1994.

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

[35] eva Kühn, R. Mordinyi, and C. Schreiber. An Extensible Space-based Coordination Approach for Modeling Complex Patterns in Large Systems. *3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Special Track on Formal Methods for Analysing and Verifying Very Large Systems*, 2008.

[36] eva Kühn, J. Riemer, R. Mordinyi, and L. Lechner. Integration of XVSM Spaces with the Web to Meet the Challenging Interaction Demands in Pervasive Scenarios. *Ubiquitous Computing And Communication Journal (UbiCC), special issue on "Coordination in Pervasive Environments"*, 3, 2008.

[37] eva Kühn, Johannes Riemer, and Geri Joskowicz. XVSM (eXtensible Virtual Shared Memory) Architecture and Application. Technical report, Space-Based Computing Group, Institute of Computer Languages, Vienna University of Technology, November 2005.

[38] Edward Fredkin. Trie Memory. *Commun. ACM*, 3(9):490–499, 1960.

[39] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[40] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[41] David Gelernter and Arthur J. Bernstein. Distributed Communication via Global Buffer. In *PODC '82: Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 10–18, New York, NY, USA, 1982. ACM.

[42] Steven L. Halter. *Javaspaces Example by Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[43] Theo Härder. DBMS Architecture - the Layer Model and its Evolution (Part I). *Datenbank-Spektrum*, 5(13):45–56, 2005.

[44] Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, and Dmitriy Kopylenko. *Professional Java Development with the Spring Framework*. Wrox Press Ltd., Birmingham, UK, UK, 2005.

[45] Gregor Kiczales and Mira Mezini. Aspect-Oriented Programming and Modular Reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM.

[46] Lukas Lechner. A JavaScript API for an eXtensible Virtual Shared Memory (XVSM). Master's thesis, Vienna University of Technology, E185/1, Space Based Computing Group, 2008.

[47] Robert Liguori and Patricia Liguori. *Java pocket guide*. O'Reilly, 2008.

[48] Richard Monson-Haefel and David Chappell. *Java Message Service*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[49] Nirmal K. Mukhi, Ravi Konuru, and Francisco Curbera. Cooperative Middleware Specialization for Service Oriented Architectures. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 206–215, New York, NY, USA, 2004. ACM.

[50] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A Middleware for Physical and Logical Mobility. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 524, Washington, DC, USA, 2001. IEEE Computer Society.

[51] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents. *ACM Trans. Softw. Eng. Methodol.*, 15(3):279–328, 2006.

[52] Jan Newmarch. *A programmer's guide to Jini technology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.

[53] Jan Newmarch. *Foundations of Jini 2 Programming*. Apress, Berkely, CA, USA, 2006.

[54] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[55] Penny Nii. The blackboard model of problem solving. *AI Mag.*, 7(2):38–53, 1986.

[56] Scott Oaks and Henry Wong. *Jini in a nutshell: a desktop quick reference*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[57] Andrea Omicini and Franco Zambonelli. Coordination of Mobile Agents for Information Systems: the TuCSoN Model. In *AI*IA'98 Workshop on Knowledge Integration*, pages 94–98, Padova, Italy, 23–25 September 1998. Edizioni Progetto Padova.

[58] Andrea Omicini and Franco Zambonelli. TuCSoN: a Coordination model for Mobile Information Agents. In David G. Schwartz, Monica Divitini, and Terje Brasethvik, editors, *1st International Workshop on Innovative Internet Information Systems (IIIS'98)*, pages 177–187, Pisa, Italy, 8–9 June 1998. IDI – NTNU, Trondheim (Norway).

[59] Gian Pietro Picco, Davide Balzarotti, and Paolo Costa. LIGHTS: A Lightweight, Customizable Tuple Space Supporting Context-Aware Applications. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 413–419, New York, NY, USA, 2005. ACM.

[60] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda Meets Mobility. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 368–377, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[61] Robert Pitschadell. Design and Implementation of an extended, serverless Java Messaging System JMS using the XVSM Shared Data Space. Master's thesis, Vienna University of Technology, E185/1, Space Based Computing Group, 2008, in work.

[62] Michael Prostler. Design and Implementation of MozartSpaces, the Java Reference Implementation of XVSM. Master's thesis, Vienna University of Technology, E185/1, Space Based Computing Group, 2008, submitted.

[63] Derek T. Sanders, Jr. J. A. Hamilton, and Richard A. MacDonald. Supporting A Service-Oriented Architecture. In *SpringSim '08: Proceedings of the 2008 Spring simulation multiconference*, pages 325–334, New York, NY, USA, 2008. ACM.

[64] Thomas Scheller. Design and Implementation of XcoSpaces, the .Net Reference Implementation of XVSM - Core Architecture and Aspects. Master's thesis, Vienna University of Technology, E185/1, Space Based Computing Group, 2008.

[65] 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, E185/1, 2008, submitted.

[66] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[67] Robert Tolksdorf and Dirk Glaubitz. XMLSpaces for Coordination in Web-Based Systems. In *WETICE '01: Proceedings of the 10th IEEE International Workshops on Enabling Technologies*, pages 322–327, Washington, DC, USA, 2001. IEEE Computer Society.

[68] Robert Tolksdorf, Franziska Liebsch, and Duc Minh Nguyen. XMLSpaces.NET: An Extensible Tuplespace as XML Middleware. In *In Report B 03-08, Free University Berlin, ftp://ftp.inf.fu-berlin.de/pub/reports/tr-b-0308.pdf, 2003. Open Research Questions in SOA 5-25 and Loose Coupling in Service Oriented Architectures*, 2004.

[69] R.A. van der Goot. *High Performance Linda using a Class Library*. PhD thesis, Erasmus University Rotterdam, 2001.

[70] Roel van der Goot, Jonathan Schaeffer, and Gregory V. Wilson. Safer Tuple Spaces. In *COORDINATION '97: Proceedings of the Second International Conference on Coordination Languages and Models*, pages 289–301, London, UK, 1997. Springer-Verlag.

[71] Jim Waldo. *The Jini Specifications*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[72] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.

[73] Michael Wittmann. XVSM Tutorial and Application Scenarios. Master's thesis, Vienna University of Technology, E185/1, Space Based Computing Group, 2008.

[74] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Syst. J.*, 37(3):454–474, 1998.

Appendix A

Jini Services

In the following, an overview about the default Jini service implementations shipped together with the Jini starter kit is given.

Reggie Reggie is the default Sun-implementation of the Jini lookup service. The lookup service is the central registry of services, where clients and services can find a service of interest [71].

Fiddler Fiddler is the default Sun-implementation of the Jini lookup-discovery service. It is a helper service, employing the Jini discovery protocol to find lookup services in which clients or services have expressed interest [71].

Mahalo Mahalo is the default Sun-implementation of the Jini transaction managers. It supports the Jini specified two-phase commit protocol (see section 3.6.1) as well as nested transactions.

Mercury Mercury is the default Sun-implementation of the Jini event mailbox service. This service purpose is to collect events on behalf of the clients and to forward them at the clients' request.

Norm Norm is the default Sun-implementation of the Jini lease renewal service. This service purpose is to renew leases of resources, before they are expiring, on behalf of the clients.

Outrigger Outrigger is the default Sun-implementation of the JavaSpaces service (see chapter 4) providing the JavaSpace05 interface (see section 4.7.2) to its clients.

Phoenix Phoenix is the default Sun-implementation of the Jini activation service. Activatable services, which are registered at the phoenix service, are inactive as long as they are not needed, in order to save resources, and will be activated as soon as the service is requested by a client.

Appendix B

JAXS - Test Cases

This appendix contains a survey of the test cases that were performed to test the compliance of the JavaSpaces API for XVSM-space (JAXS) implementation with the JavaSpaces specification.

The following assumptions are made, if not explicitly otherwise stated, for all test cases:

- An HTTP server, a Jini lookup service (e.g. Reggie) and a Jini transaction service (e.g. Mahalo) are running.
- All tests are performed without transactions, with a maximum lease duration and a 5 seconds timeout.
- Only the provided methods of the JavaSpace interface are used to access the space.

To be able to perform the test cases, the Jini `Entry` class has been implemented by the `SimpleEntry` class, which was further extended by the `ExtendedEntry` class. The ExtendedEntry class is required for testing the subtype matching mechanism of the read, take and notify operations. Both classes are depicted in figure B.1.

B.1 Service Registration and Lookup

The following test cases (depicted in table B.1) test the registration and the lookup of the JAXS service.

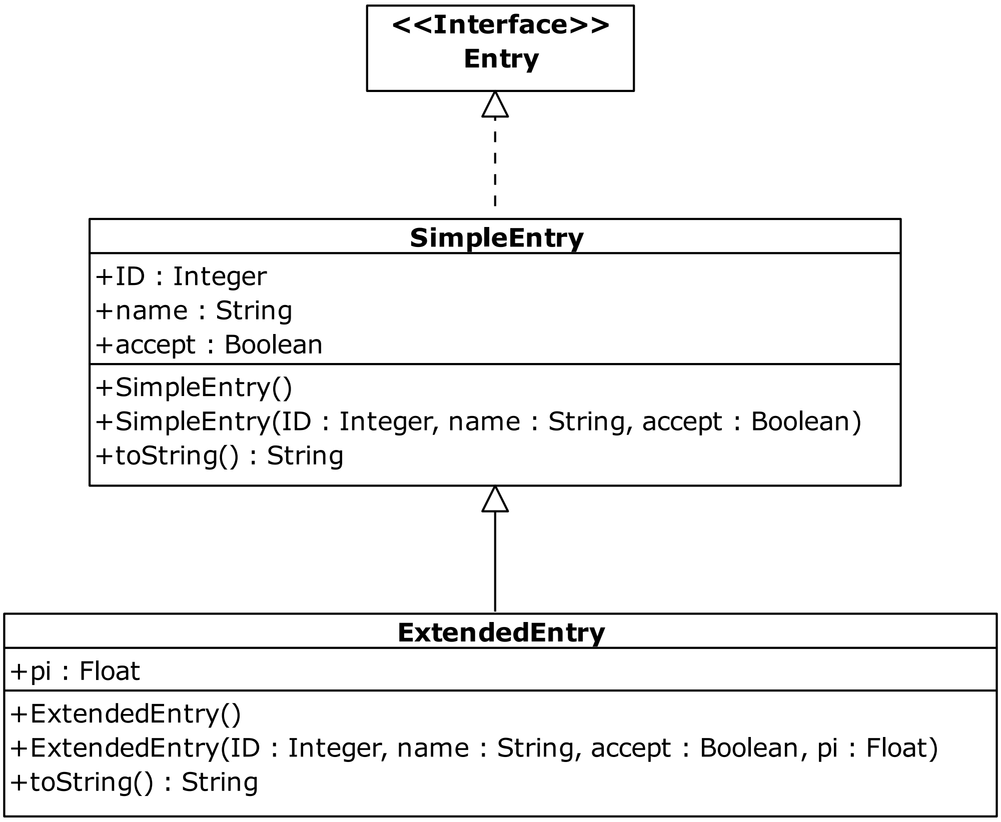


Figure B.1: SimpleEntry and ExtendedEntry classes

ID	**Description**	**Precondition**	**Expected Result**
S.1	Start JAXS to register it at a lookup service.	Only a HTTP server, a lookup service and a transaction service are running.	JAXS is registered at the lookup service.
S.2	Lookup JAXS service using the JavaSpace interface.	JAXS service is running.	Receive a reference to the JAXS service.
S.3	Lookup JAXS service using the JavaSpace05 interface.	JAXS service is running.	Receive a reference to the JAXS service.
S.4	Lookup JAXS service using the IXVSMService interface.	JAXS service is running.	Receive a reference to the JAXS service.
S.5	Shutdown JAXS and restart it again	JAXS service is running.	The JAXS service is registered at the lookup service once only.

Table B.1: Service registration and lookup test cases

B.2 JavaSpace Operations

The following test cases (depicted in table B.2) test the basic operations of the JavaSpace interface as specified by JavaSpaces. These tests also include test cases for subtype matching of entries, as specified by the Jini entry specification [71]. The *notification* method is omitted at this point and will be tested later in section B.5.

ID	Description	Precondition	Expected Result
J.1	Writing a SimpleEntry	Empty space.	The space contains the written entry and a Lease object is returned by JAXS.
J.2	Reading with a blank template of type SimpleEntry.	Empty space.	The read operation clocks until its timeout expires. Afterwards an exception is thrown.
J.3	Reading with a blank template of type SimpleEntry.	10 SimpleEntries in the space.	Return of a single matching SimpleEntry.
J.4	Reading with a partially filled template of the type SimpleEntry.	10 SimpleEntries in the space.	Return of a single matching SimpleEntry.
J.5	Reading with a blank template of type SimpleEntry. (Subtype-Matching)	10 entries of each type in the space.	Return of a single matching entry either of type SimpleEntry or ExtendedEntry.
J.6	Reading with a blank template of type ExtendedEntry. (Subtype-Matching)	10 entries of each type in the space.	Return of a single entry of type ExtendedEntry.
J.7	Reading with a blank template of type ExtendedEntry. (Subtype-Matching)	10 SimpleEntries in the space.	The read operation clocks until its timeout expires. Afterwards an exception is thrown.
J.8	Taking with a blank template of type SimpleEntry.	10 entries of each type in the space.	Return of a matching SimpleEntry and its removal from the space.

continued on next page

continued from previous page			
ID	**Description**	**Precondition**	**Expected Result**
J.9	Consecutive taking with an entry created with the *snapshot* method.	10 entries of each type in the space. Snapshot of a blank template of type SimpleEntry.	Return of all entries and their removal from the space.
J.10	Invocation of the *readIfExist* method with a blank template of type SimpleEntry.	Empty space.	Immediate return without a result.
J.11	Invocation of the *readIfExist* method with a blank template of type SimpleEntry. Rollback of the open transaction	A single SimpleEntry in the space, which is unavailable because it is taken by an unsettled transaction.	The *readIfExist* operation must block until the transactional state of the potentially matching entry is decided. In this case, it returns the matching entry
J.12	Invocation of the *takeIfExist* method with a blank template of type SimpleEntry.	Empty space.	Immediate return without any results.
J.13	Invocation of the *takeIfExist* method with a blank template of type SimpleEntry. Rollback of the open transaction	A single SimpleEntry in the space, wich is unavailable because it is taken by an unsettled transaction.	The *takeIfExist* operation must block until the transactional state of the potentially entry is decided. In this case, it returns the matching entry.

Table B.2: Basic operation test cases

B.3 Javaspace05 Operations

The following test cases (depicted in table B.3) test the basic operations of the JavaSpace05 interface as specified by JavaSpaces. The *registerForAvailabilityEvent* method is omitted at this point and will be tested later in section B.5.

ID	**Description**	**Precondition**	**Expected Result**
E.1	Invocation of the *contents* method with a blank template of type SimpleEntry.	10 entries of each type in the space	Return of a MatchSet object that iterates over all matching entries.
E.2	Invocation of the *contents* method with a blank template of type SimpleEntry and afterwards invalidation through a *take* operation	10 entries of each type in the space.	Return of a MatchSet object that is invalidated after the performed *take* operation.
E.3	Invocation of the *contents* method with 3 differently filled templates of type SimpleEntry, which will match more than one entry in the space.	10 entries of each type in the space	Return of a MatchSet object that iterates over all matching entries.
E.4	Writing 5 entries of each type at once with the *JavaSpace.write* method.	Empty space.	The space contains all written entries and a Lease object is returned for each written entry.
E.5	Try to take 10 entries with the *JavaSpace.take* method and a blank template of type ExtendedEntry.	10 entries of each type in the space	Return of 10 matching ExtendedEntries.

Table B.3: JavaSpace05 interface test cases

B.4 Transaction

The following test cases (depicted in table B.4) test the correct behavior of JAXS' operations under transactions as specified by JavaSpaces. The support of the 2-phase commit protocol (see section 3.6.1) is not tested explicitly. The correct implementation of the 2-phase commit protocol is rather assumed from the results of the test cases performed with one or more clients.

ID	**Description**	**Precondition**	**Expected Result**
T.1	Write a SimpleEntry under a new transaction. Commit the transaction afterwards.	Empty space.	The space contains the written entry.
T.2	Write a SimpleEntry under a new transaction. Rollback the transaction afterwards.	Empty space.	The space is still empty.
T.3	Perform a take operation with a blank template of type SimpleEntry under a new transaction. Commit the transaction afterwards.	10 SimpleEntries in the space.	Return of a single matching entry. The entry is removed from the space.
T.4	Perform a take operation with a blank template of type SimpleEntry under a new transaction. Rollback the transaction afterwards.	10 SimpleEntries in the space.	Return of a single matching entry and the space still containes the entry.
T.5	Use a newly created transaction to write a SimpleEntry and use the same transaction to take the entry again. Commit the transaction.	Empty space.	The space is still empty.
T.6	Use a newly created transaction to write a SimpleEntry and use the same transaction to take the entry again. Rollback the transaction.	Empty space.	The space is still empty.

continued on next page

continued from previous page

ID	**Description**	**Precondition**	**Expected Result**
T.7	Create a transaction with a 5 seconds timeout. Perform a take operation with a blank template of type SimpleEntry under the previously created transaction. Create another transaction with maximum timeout and perform a take with a blank template of type SimpleEntry under the new transaction.	Empty space.	The second take operation must block until the first transaction is rolledback automatically after the timeout and return a single entry.
T.8	Create a transaction with a 5 seconds timeout. Write a SimpleEntry under the previously created transaction. Create another transaction with maximum timeout and perform a take with a blank template of type SimpleEntry under the new transaction.	Empty space.	The second take operation must block until its timeout expires.
T.9	Create a transaction with a 5 seconds timeout. Write a SimpleEntry under the previously created transaction. Perform a take with a blank template of type SimpleEntry again.	Empty space.	The second take operation must block until its timeout expires.
T.10	Create a transaction with a maximum timeout. Perform a take operation with a blank template of type SimpleEntry under the previously created transaction. Use another thread or client to create another transaction and perform a write operation with an SimpleEntry under the lastly created transaction. Afterwards rollback the last transaction.	Empty space.	The first take operation must block until its timeout expires.

continued on next page

continued from previous page

ID	**Description**	**Precondition**	**Expected Result**
T.11	Create a transaction with a maximum timeout. Perform a take operation with a blank template of type SimpleEntry under the previously created transaction. Use another thread or client to create another transaction and perform both a write operation with an SimpleEntry and a take operation with a blank template of type SimpleEntry under the lastly created transaction. Afterwards commit the last transaction.	Empty space.	The first take operation must block until its timeout expires.

Table B.4: Transaction test cases

B.5 Notification

The following test cases (depicted in table B.4) test the correct behavior of JAXS' notifications as specified by JavaSpaces. The test cases for the method *JavaSpace.registerForAvailabilityEvent* are invoked, if not otherwise stated, with the parameter *visibilityOnly* set to "true".

ID	**Description**	**Precondition**	**Expected Result**
N.1	Create a notification with a blank template of type SimpleEntry. Write a new SimpleEntry.	Empty space.	A notification is received immediately after writing the entry.
N.2	Create a notification with a blank template of type SimpleEntry. Write a new ExtendedEntry. (Subtype-Matching)	Empty space.	A notification is received immediately after writing the entry.

continued on next page

continued from previous page

ID	Description	Precondition	Expected Result
N.3	Create a notification with a blank template of type ExtendedEntry. Write a new SimpleEntry. (Subtype-Matching)	Empty space.	No notification is expected.
N.4	Create a notification with a blank template of type ExtendedEntry. Write a new ExtendedEntry. (Subtype-Matching)	Empty space.	A notification is received immediately after writing the entry.
N.5	Create a notification with a partial matching template of type SimpleEntry. Write 5 different potentially matching entries of type ExtendedEntry.	Empty space.	A notification is received after each written entry.
N.6	Create a notification with a partial matching template of type SimpleEntry. Write an entry that does not contain any matching content.	Empty space.	No notification is expected.
N.7	Based on test case N.6, create 2 notifications each for a seperate client. Write 5 different potentially matching entries of type ExtendedEntry.	Empty space 2 seperate clients.	Each client receives a notification after each written entry. The notification ID and sequence number must be the same for both clients, but the order inwhich the notifications are received can be different at each client.

continued on next page

continued from previous page

ID	Description	Precondition	Expected Result
N.8	Create a notification using the *JavaSpace05.registerForAvailabilityEvent* method with 3 different templates: 1) A blank template of type SimpleEntry. 2) 2 differently and partially filled templates of type ExtendedEntry. Also set the boolean visibilityOnly to false. Write 5 different potentially matching entries of type ExtendedEntry.	Empty space.	A single notification should be received after each written entry, regardless of how many templates are matching.
N.9	Create a notification using the *JavaSpace05.-registerForAvailabilityEvent* method with a blank template of type SimpleEntry. Use a transaction to take an entry with a blank template of type SimpleEntry and rollback the transaction afterwards.	10 SimpleEntries in the space.	A single notification should be received after the rollback of the transaction.
N.10	Based on test case N.8, create a notification for a each client. Write 5 different potentially matching entries of type ExtendedEntry.	Empty space 2 seperate clients.	Each client receives only a single notification for each written entry, regardless of how many templates are matching. The notification ID and sequence number must be the same for both clients, but the order inwhich the notifications are received can be different at each client.

continued on next page

continued from previous page

ID	Description	Precondition	Expected Result
N.11	Based on test case N.6 and N.8, create a notification for each test case for each client. Write 5 different potentially matching entries of type ExtendedEntry.	Empty space 2 seperate clients.	Each client receives two notifications (one for each notification type) for each written entry. The notification ID and sequence number must be the same for both clients and unique with respect to each notification type. The order inwhich the notifications are received can be different at each client.

Table B.5: Notification test cases

B.6 Lease

The following test cases (depicted in table B.4) test the correct behavior of JAXS' leases as specified by JavaSpaces. The differentiation between the notification methods *notify* and *registerForAvailabilityEvent* is omitted because the test cases are the same for both types.

ID	Description	Precondition	Expected Result
L.1	Write a SimpleEntry with a lease duration of 10 seconds. Wait 5 seconds and perform a read operation with a blank template of type SimpleEntry. Wait 10 seconds and perform a read operation with a blank template of type SimpleEntry.	Empty space.	The first read operation returns the entry, whereas the second read operation blocks and throws an exception after its timeout.

continued on next page

continued from previous page

ID	**Description**	**Precondition**	**Expected Result**
L.2	Write a SimpleEntry with a lease duration of 10 seconds. Renew the lease every 5 second and perform a read operation shortly after each invocation of *renew*.	Empty space.	The entry is available as long as the lease is renewed periodically.
L.3	Write a SimpleEntry with a lease duration of 60 seconds. Wait 5 seconds and perform a read operation with a blank template of type SimpleEntry. Shortly afterwards cancel the lease. Wait another 5 seconds and perform a read operation with a blank template of type SimpleEntry again.	Empty space.	The first read operation returns the entry, whereas the second read operation blocks and throws an exception after its timeout.
L.4	Create a notification with a lease duration of 10 seconds and a blank template of type SimpleEntry. Wait 5 seconds and write a SimpleEntry. Wait another 10 seconds and write another SimpleEntry.	Empty space.	Shortly after the first write operation a notification is received, whereas the second write operation does not trigger any notification.
L.5	Create a notification with a lease duration of 60 seconds and a blank template of type SimpleEntry. Wait 5 seconds and write a SimpleEntry. Shortly afterwards cancel the lease. Wait another 5 seconds and write a SimpleEntry again.	Empty space.	Shortly after the first write operation a notification is received, whereas the second write operation does not trigger any notification.

continued on next page

continued from previous page			
ID	**Description**	**Precondition**	**Expected Result**
L.6	Create a notification with a lease duration of 10 seconds and a blank template of type SimpleEntry. Renew the lease every 5 second and write a SimpleEntry shortly after each invocation of *renew*.	Empty space.	The notifications are as long received as the lease is renewed periodically.
L.7	Create a MatchSet with the *JavaSpace05.contents* method, a lease duration of 10 seconds and a blank template of type SimpleEntry. Wait 5 seconds and invoke the *MatchSet.next* method. Wait 10 seconds and invoke the *MatchSet.next* method again.	10 entries of each type in the space	The first invocation the *MatchSet.next* method returns an entry, whereas the second invocation throws an exception indicating that the MatchSet is invalidated.
L.8	Create a MatchSet with the *JavaSpace05.contents* method, a lease duration of 10 seconds and a blank template of type SimpleEntry. Renew the lease every 5 second and invoke the *MatchSet.next* method shortly after each renew.	10 entries of each type in the space	The MatchSet is iterated as long as the lease is renewed periodically and as long as there are entries left in the MatchSet.
L.9	Create a MatchSet with the *JavaSpace05.contents* method, a lease duration of 60 seconds and a blank template of type SimpleEntry. Wait 5 seconds and invoke the *MatchSet.next* method. Shortly afterwards cancel the lease. Wait another 5 seconds and invoke the *MatchSet.next* method again.	10 entries of each type in the space	The first invocation the *MatchSet.next* method returns an entry, whereas the second invocation throws an exception indicating that the MatchSet is invalidated.

Table B.6: Lease test cases