TripCom

Triple Space Communication

FP6 – 027324

Deliverable

D6.3

Platform API Specification for Interaction Between All Components

Lyndon JB Nixon (FUB)
Daniel Martin (USTUTT)
Daniel Wutke (USTUTT)
Martin Murth (TUW)
Elena Simperl (LFUI)
Reto Krummenacher (LFUI)
Brahmananda Sapkota (NUIG)
Zhangbing Zhou (NUIG)
Hans Moritsch (TUW)
Christian Schreiber (TUW)
Omair Shafiq (LFUI)
Germán Toro del Valle (Telefonica)
Davide Cerri (CEFRIEL)
Vassil Momtchev (ONTO)

Executive Summary

The goal of this deliverable is to present the Triple Space architecture as realised in the first TripCom implementation. In particular, as the chosen architecture is component based, using a tuplespace for intercomponent communication as was given in [6], we will define how the components of the Triple Space implementation will communicate to one another in order to realise the complete functionality of the system. We see a number of different aspects in this communication which will be referred to in this deliverable:

- Use of a space implementation as the integrating middleware / service bus of the Triple Space kernel
- Contents of the messages to be passed from one component to another in the service bus
- Workflow of the communication to ensure active processes in a kernel are handled as efficiently as possible

This will be the focus of Chapter 3 of this deliverable. Prior to that, we develop a basis for determining the specification of the component interaction based on the coordination patterns supported by the Triple Space system and the definition of the intended functionality of each component in the architecture (Chapter 2). To start, we define terms used in the text in a glossary and specify explicitly assumptions being made in this work (Sections 1.2 and 1.3). The described architecture and component integration is realised in the first TripCom implementation, which acts as a proof of the work and a testbed for scalability and further extensions planned for the next and final year of the TripCom project. The prototype is described in Chapter 4 and in Chapter 5 we conclude with a summary of our plans for the second TripCom implementation, which is the final output of the project.

Document Information

IST Project Number	FP6 – 027324	**Acronym**	TripCom
Full Title	Triple Space Communication		
Project URL	http://www.tripcom.org/		
Document URL			
EU Project Officer	Werner Janusch		

Deliverable	**Number**	6.3	**Title**	Platform API Specification for Interaction Between All Components
Work Package	**Number**	6	**Title**	Triple Space Architecture and Component Integration

Date of Delivery	**Contractual**	M24	**Actual**	31-March-08
Status	version 1.0		final ☒	
Nature	prototype ☐ report ☒ dissemination ☐			
Dissemination Level	public ☒ consortium ☐			

Authors (Partner)	Lyndon JB Nixon			
Resp. Author	Lyndon JB Nixon		**E-mail**	nixon@inf.fu-berlin.de
	Partner	FUB (Free University of Berlin)	**Phone**	(xxx) xxxx-xxx

Abstract (for dissemination)	This deliverable presents the Triple Space architecture as it has been implemented in the first TripCom implementation. In particular, as the chosen architecture is component based and uses a tuplespace for intercomponent communication, we specify how the individual components of a Triple Space kernel communicate with one another in order to realise the complete functionality of the system.
Keywords	Triple Space, components, kernel, architecture, interaction, integration

Version Log			
Issue Date	**Rev No.**	**Author**	**Change**
2007-05-15	1	Daniel Martin	First draft structure
2007-05-25	2	Lyndon Nixon	First descriptions of sections
2007-06-06	3	Daniel Martin	Second draft structure
2007-06-22	4	Lyndon Nixon	Second descriptions of sections
2007-07-02	5	Daniel Martin	Component description template update
2007-07-11	6	Elena Simperl	Development process first draft
2007-07-12	7	Martin Murth	Component interaction diagram first draft
2007-07-10	8	Daniel Martin	Component interaction diagram update
2007-07-17	9	Brahmananda Sapkota	Deployment architecture first draft
2007-07-22	10	Zhangbing Zhou	Deployment architecture initial update
2007-07-30	11	Zhangbing Zhou	Component descriptions for discovery and invocation first draft
2007-07-31	12	Daniel Martin	Component interaction diagram first draft
2007-07-31	13	Lyndon Nixon	TS API Clients and Operations first draft
2007-07-31	14	Brahmananda Sapkota	Deployment architecture first draft update
2007-07-31	15	Hans Moritsch	Added text to activity diag
2007-08-01	16	Omair Shafiq	Updated section, Web Services clients API
2007-08-21	17	Elena Simperl	Updated section on architecture development process
2007-09-17	18	Zhangbing Zhou	Updated section for discovery and invocation
2007-09-25	19	Lyndon Nixon	Component descriptions for TS API, Management API, Distribution Manager, Query Processor and Transaction Manager
2007-10-15	20	Vassil Momtchev	Component description for Triple Store Adapter
2007-10-25	21	Lyndon Nixon	Added goals and assumptions of TripCom
2007-10-30	22	Daniel Martin	extended and planned section on XVSM extensions
2008-01-18	23	Lyndon Nixon	new structure in line with WP6 activities
2008-03-06	24	Daniel Martin	revised sections assigned to USTUTT
2008-03-10	25	Hans Moritsch, Christian Schreiber	revised component integration chapter
2008-03-13	26	Lyndon Nixon	revised FUB sections, esp. replaced API workflows with revised API tables
2008-03-21	27	Christian Schreiber, Hans Moritsch	new chapter on triple space prototype
2008-04-01	28	Hans Moritsch, Lyndon Nixon	Revisions following quality review by WP leader

Project Consortium Information

Acronym	Partner	Contact
Semantic Technology Institute Innsbruck http://www.sti-innsbruck.at	STI	Prof. Dr. Dieter Fensel Semantic Technology Institute (STI) Innsbruck, Austria E-mail: dieter.fensel@sti-innsbruck.at
National University of Ireland, Galway http://www.deri.ie	NUIG	Dr. Laurentiu Vasiliu Digital Enterprise Research Institute (DERI) Galway, Ireland Email: laurentiu.vasiliu@deri.org
University of Stuttgart http://www.iaas.uni-stuttgart.de/	USTUTT	Prof.Dr. Frank Leymann Inst. für Architektur von Anwendungssystemen (IAAS) Stuttgart, Germany E-mail: frank.leymann@informatik.uni-stuttgart.de
Vienna university of Technology http://www.complang.tuwien.ac.at/	TUW	Prof.Dr. eva Kühn Institut für Computersprachen Vienna, Austria E-mail: eva@complang.tuwien.ac.at
Free University Berlin http://www.ag-nbi.de/	FUB	Prof. Dr.-Ing. Robert Tolksdorf AG Netzbasierte Informationssysteme Berlin, Germany E-mail : tolk@inf.fu-berlin.de
Ontotext Lab, Sirma Group Corp. http://www.ontotext.com/	ONTO	Atanas Kiryakov, Vassil Momtchev, Ontotext Lab, Sirma Group Corp. Sofia, Bulgaria E-mail: vassil.momtchev@ontotext.com
Profium OY http://www.profium.com/	Profium	Dr. Janne Saarela Profium OY Espoo, Finland E-mail: janne.saarela@profium.com
CEFRIEL SCRL. http://www.cefriel.it/	CEFRIEL	Davide Cerri CEFRIEL SCRL. Milano, Italy E-mail: cerri@cefriel.it
Telefonica I+D http://www.tid.es/	TID	Noelia Pérez Crespo Telefonica I+D Madrid, España E-mail: npc@tid.es

Table of Contents

1 Introduction 2
1.1 Goals . . . 2
1.2 Glossary . . . 2
1.3 Assumptions in the work . . . 4
1.3.1 Triple Space configurations . . . 4
1.3.2 Triple Space vs triplespace . . . 4
1.3.3 Spaces vs data . . . 5
1.3.4 RDF vs WSML . . . 5
1.3.5 Triple Space URIs and URLs . . . 5

2 Triple Space Architecture 7
2.1 Overview . . . 7
2.1.1 Architectural Views . . . 8
2.1.2 The Triple Space Architecture . . . 8
2.2 Coordination Patterns . . . 10
2.2.1 Management API . . . 13
2.3 Component Descriptions . . . 14
2.3.1 Triple Space API . . . 14
2.3.2 Management API . . . 15
2.3.3 Triple Store Adapter . . . 17
2.3.4 Query Pre-Processor . . . 21
2.3.5 Distribution Manager . . . 22
2.3.6 Metadata Manager . . . 26
2.3.7 Transaction Manager . . . 29
2.3.8 Security Manager . . . 32
2.3.9 Web Service Registry . . . 33
2.3.10 Web Service Discovery . . . 36
2.3.11 Web Service Invocation . . . 36
2.3.12 Mediation Manager . . . 39

3 Component Integration 42
3.1 Integration Middleware . . . 42
3.1.1 JavaSpaces . . . 43
3.1.2 Blitz Implementation . . . 44
3.2 Implementation Plan . . . 44
3.3 Development Tools and Platform . . . 45
3.4 Component Interfaces . . . 46
3.4.1 Entry descriptions . . . 48

4 First TripCom Implementation 54
4.1 Starting and deploying the prototype . . . 54
4.1.1 Required Software . . . 54
4.1.2 Dependencies . . . 54
4.2 Blitz . . . 55
4.2.1 Configuration . . . 55
4.2.2 Starting Blitz . . . 56

4.3 Compile 56
4.3.1 Obtaining the source 56
4.3.2 Compiling the source 57
4.4 Example 57
4.5 Realized functionality 59
4.6 Statistics 59

5 Conclusion 62

List of Abbreviations

API	Application Programming Interface
ASD	Agile Software Development
CVS	Concurrent Versioning System
CMM	Capability Maturity Model
COCOMO	Constructive Cost Model
CSF	Critical Success Factor
DCM	Digital Contents Management
DoW	Description of Work, Tripcom Annex I.
EC	European Commission
EDI	Electronic Data Interchange
EU	European Union
UN/EDIFACT	United Nations EDI For Admin., Commerce and Transport
MEP	Message Exchange Pattern
OWL	Web Ontology Language
OWL-S	Semantic Markup for Web Services
P2P	Peer to Peer
PKI	Public Key Infrastructure
PSP	Personal Software Process
RDF	Resource Description Framework
RDFS	RDF Schema
RUP	Rational Unified Process
SVN	Subversion
SW-CMM	Software CMM
TripCom	Triple Space Communication
TSC	Triple Space Computing
TS API	Triple Space API
TSP	Team Software Process
UC	Use Case
UML	Unified Modelling Language
UP	Unified Process
W3C	World Wide Web Consortium
WP	Work Package
WS	Web Service
WSDL	Web Service Description Language
WSML	Web Service Modelling Language
WSMT	Web Service Modelling Toolkit
WSMX	Web Service Execution Environment
XML	Extensible Mark-up Language

1 Introduction

1.1 Goals

The goal of this deliverable is to present the Triple Space architecture as implemented in the first TripCom implementation. In particular, as the chosen architecture is component based using a tuplespace for intercomponent communication as was given in [6], we will define how the components of the Triple Space implementation are coordinated with one another in order to realise the complete functionality of the system. We see a number of different aspects of coordination which will be referred to in this deliverable:

- Use of a space implementation as the integrating middleware / system bus of the Triple Space kernel
- Contents of the messages to be passed from one component to another via the service bus
- Kernel internal workflow to ensure active processes are handled as efficiently as possible

This will be the focus of Chapter 3 of this deliverable. Prior to that, we develop a basis for determining the specification of the component interaction based on the coordination patterns supported by the Triple Space system and the definition of the intended functionality of each component in the architecture (Chapter 2). To start, we define terms used in the text in a glossary and specify explicitly assumptions being made in this work (Sections 1.2 and 1.3). The described architecture and component integration is implemented in a second TripCom prototype, which acts as a proof of concept and a testbed for scalability evaluation and further extensions planned for the next and final year of the TripCom project. The prototype is described in Chapter 4 and in Chapter 5 we conclude with a summary of our plans for the third prototype, which is the final implementation output of TripCom.

1.2 Glossary

Let us first define the terminology used in the rest of this document, as an update of the glossary first provided in the deliverable D6.2:

Throughout this document we use the terms listed below to describe the concepts of the Triple Space architecture.

Triple Space (TS) Triple Space denotes the infrastructure which will be developed in TripCom.

tuplespace A tuplespace is a shared associative memory whose elementary data units are tuples. A tuple is a heterogeneous, ordered collection of typed values.

triplespace A triplespace is a tuplespace in which every tuple represents an RDF statement in terms of a **triple** `<subject, predicate, object>`. A triplespace is identified by a unique identifier, i.e., the name of the triplespace. A triplespace is accessed via the Triple Space API primitives.

subspace A triplespace can contain subspaces. A subspace is a part of a triplespace that complies to the definition of a triplespace and can contain subspaces itself. If a triple is in two spaces, then one of them must be a subspace of the other, i.e., subspaces cannot arbitrarily overlap. A space which is not a subspace is termed a **root space**.

kernel A kernel is the sum of software components ("kernel components"), which, as a whole, provide under a single address and authority the functionality of Triple Space, up to a certain extent. This extent is to be specified in terms of a Triple Space kernel configuration.[1] A particular execution of a kernel is termed a **kernel instance**.

kernel group A kernel group is the set of kernels instances that manage a particular triplespace, or any of its subspaces. The kernel group is said to manage this triplespace. It is also termed the (managing) kernel group *of* this triplespace. A kernel instance, as well as a kernel group, can manage more than one space.

kernel component (component) A kernel component is a self-contained part of a kernel which provides a specific functionality and has a well-defined interface to other kernel components. The particular execution of a kernel component is termed a **component instance**.

Triple Space features two kinds of distribution. A space can be physically distributed (*space distribution*), and a kernel can be executed in a distributed manner (*kernel distribution*). Correspondingly, the following abstractions originate.

partition A triplespace can be physically distributed, i.e., every triple is assigned to a set of kernel instances (which is a subset of the managing kernel group of that space). A distribution function maps triples to kernel instances. As a result, the space decomposes into disjoint partitions in such a way that all triples within a partition are assigned to the same set of kernel instances.

One single partition assigned to a single kernel instance corresponds to a space which is not distributed. Triples assigned to more than one kernel instance are replicated.[2]

As the distribution of a space is based on triples themselves, it is orthogonal to its structuring into subspaces. The set of kernel instances to which a triple is assigned (which is equivalent to the partition it belongs to) is independent of its membership to subspaces. Due to the properties of subspaces, the distribution of a root space implicitly specifies the distribution of its subspaces.

kernel processor A kernel component is executed on a physical machine which is addressable through its network address. The kernel as a whole can be executed on one ore more physical machines.

[1] A kernel configuration is a vector of (functional and non-functional) kernel properties.

[2] The fact that a particular triple is not assigned to a kernel instance according to a defined distribution does not imply that no physical replication of that triple may exist at the machine(s) running that kernel instance. Rather, replications may exist for certain purposes which are not captured by the specified mapping. Such replications may define responsibilities within an implementation.

A kernel can be executed in a distributed manner by executing components on multiple machines, i.e., through distributing the component instances establishing a kernel instance—potentially including multiple instances of the same component—onto a set of physical machines. This set is referred to as the (kernel) processor which runs the kernel instance.

In case of a non-distributed kernel execution, the kernel processor is identical to the single machine on which all component instances are executed.

1.3 Assumptions in the work

In the context of the first TripCom implementation, it is also important to ensure that all assumptions and goals of the work are explicit as these (often then implicitly) guide the implementation decisions that are made. All components, which make up in totality the Triple Space kernel, must be designed with these goals and assumptions in mind.

1.3.1 Triple Space configurations

Triple Space will support different configurations. The purpose of these configurations is to provide different trade-offs with respect to scalability of the system and the expressiveness of the co-ordination model it supports. Fundamentally, a core Triple Space configuration is specified which all components must support (another issue is that some components may not be necessary in such a configuration) and which has the aim to support Web-like scalability. Further extensions/limitations of the core configuration will be specified to support specific scenarios. Those scenarios will then make the requirement to operate only with kernels which are able to support the configuration that they use. Components may only optionally support configurations beyond the core configuration, i.e. there is no obligation that a component implementation support a certain non-core configuration. Rather, implementations will aim to be as flexible as possible with respect to configurations, and configurations which include functionalities for which an implementation can not be provided may need to fall back onto the core configuration to ensure that they function in Triple Space. In TripCom, two further configurations will be specified: one for an eHealth scenario (European Patient Summary) and one for an EAI scenario (Digital Asset Marketplace). These will give component implementers the opportunity to test their components for supporting non-core configurations.

1.3.2 Triple Space vs triplespace

Just as one can differentiate between a single Web server and the World Wide Web, one needs to be able to differentiate between a triplespace (which is managed by a kernel) and the Triple Space which is the global sum of all triplespaces managed by kernels at one time. Whether client interactions can be directed to the Triple Space in its entirety or are restricted to named triplespaces has an impact on how components function. Generally, this is controlled by whether or not a triplespace identifier (generally an URL) is provided in an API call to a kernel. The core configuration will support read operations (i.e. non-destructive retrieval) on the Triple Space, and support for out, in (destructive retrieval) or subscribe operations will be restricted to named spaces.

1.3.3 Spaces vs data

The Triple Space system, as implemented in a software bundle called a kernel, can manage and organize both (triple)spaces and data (which belongs in a space). What is permissible in terms of this organisation can have an impact on how the system can find and use spaces and data. Fundamentally, issues arise around the distribution of spaces and data. Fundamentally (in the core configuration) a space will be managed on a single kernel, and hence is findable through name resolution (space X/1 is managed on kernel X, where X is the kernel hostname). Data is placed within that space. However, in other configurations we will allow that a space may be managed across multiple kernels. When data is placed within that space, a decision can be made on which kernel that data will be managed using the Metadata Manager. The basis of that decision needs to be re-usable when data will be retrieved, so that it can be known which kernel to ask. Generally, data in a space managed across kernels can be *clustered* on kernels, i.e. data which is semantically or syntactically more similar can be placed closer together.

1.3.4 RDF vs WSML

Triple Space has been foreseen as being a RDF-based coordination platform. It has not been ruled out that other knowledge representations - based on other logical formulisms - may also be supported in extended configurations of the system. Hence it is desirable that - as far as possible - components are implemented in a way independant on the underlying knowledge representation of the data they will handle. In cases where this is understandably unavoidable, e.g. by reasoning, the component should be able to determine the knowledge representation it is dealing with and dynamically select the most appropriate sub-component from a set which provide support for different representations. However, every component can not guarantee to support any given knowledge representation beyond RDF, and every configuration should clearly define which knowledge representations should be supported in that configuration. For example, in the two extended configurations of TripCom, we expect support for WSML ontologies and instance data. Finally, where possible, more expressive knowledge representations should be supported for exchange, if not reasoning, through mappings to the RDF triple model (such as is possible with WSML).

1.3.5 Triple Space URIs and URLs

To address Triple Space kernels and spaces, we will use a commonly agreed identification scheme. For kernels, we will use the URL scheme, enabling kernels to be found using DNS. Hence kernel URLs will be determined by the usual means of domain names registered to a server, and in DNS mappable to a certain IP address. For spaces, we will use the URI scheme (as spaces are not physical components on a machine, but virtual abstractions of a tuple store).

As protocol, we propose an unofficial prefix tsc://. This overrides the use of tripcom:// in the previous deliverable D6.2. Unofficial prefixes are already in use for various communication networks such as mms://, mvn:// and skype://. We choose to express kernel and space identifiers with this prefix rather than a common and official protocol such as http:// for the following reasons:

- If we consider the http protocol specification rfc2616 [3], http is designed to publish and retrieve hypertext pages over the Internet - which means transfer of text. Considering the exchange of data using Triple Space, we could transfer our data over the Internet in other formats than text. Hence we wish to indicate through our protocol that communication is not necessarily restricted to text.

- Users noting the use of the HTTP protocol may expect that we implement the methods of http protocol, like GET, POST, PUT, TRACE in the communication model of Triple Space. We would need a mapping of these methods to our TS API operations, which is not trivial [4].

- Access to Triple Space is meant to be abstracted from the underlying transport protocol. If supported in the API, we could just as easily allow access over FTP or SMTP or any other protocol other than HTTP. There is also local access within a machine, apart from any transport protocol. Hence an abstract protocol better indicates this protocol-independence.

Hence we prefer to use tsc:// as an unofficial protocol, and keep it open to be defined later - if we really need to define it. In many cases there is no need to define it, like mms:// or mvn://. tsc:// can be realized by SOAP, RMI or socket server connects, among others.

We agree to this URI/URL scheme for spaces/kernels:

```
tsc://example.com:8080/thefirstsupcase/fuberlinspace
\ /   \______________/\______________/ \__________/
 |            |
scheme  roootspace/kernel  subspace       subspace
```

Note that the kernel URL and URI of the root space of that kernel are the same. A single kernel may have multiple root spaces (just as a web server may have multiple domain names) which may be replicates (like a website mirror) or hold separate content.

[3] http://tools.ietf.org/html/rfc2616

[4] This has been noted in http://rest.blueoxen.net/cgi-bin/wiki.pl?LindaAndTheWeb, last checked April 9 2008

2 Triple Space Architecture

2.1 Overview

In this section we introduce the approach applied to develop the prototype and how we created the architectural model for our large-scale distributed semantic space platform. Given the particularities of the project setting – the relatively short development cycles, the prototypical nature of the development activity and the relatively small size of the development team – we opted for a Scrum-based[1] development approach.

Scrum is an agile process to manage and control software development work. It is centered around so called "backlogs" (see Figure 2.1), a prioritized description of items that need to be done in order to achieve a certain goal, very much like a detailed to do list for a specific task. Backlogs come in two forms, product backlog – the to do list for the whole product, and sprint backlogs – the to do list for a specific iteration within the product development phase. A sprint typically lasts two to four weeks, with dayly scrum meetings where the development team comes together, provides status updates on the sprint backlog and discusses problems that may have occured. Further details on the development process and how this process is applied in TripCom is provided in Section 3.2.

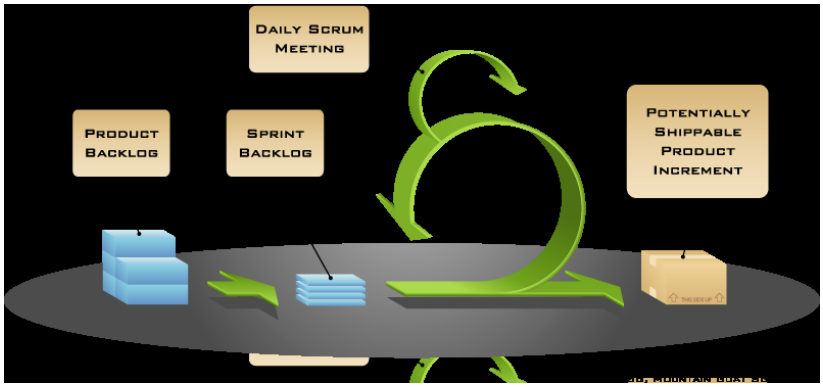

Figure 2.1: Scrum Lifecycle

On the other hand, our architectural model was defined as a sequence of phases, using an initial architecture [6] outline as input in order to decide upon core architectural issues, and then eventually produce a complete system architecture captured as a set of views. The architectural development process consists of the following phases:

Refine architectural mechanisms After a revision of the objectives of the prior iteration, architectural mechanisms are refined into an implementation state.

Identify architecturally significant design elements Key abstractions are refined into concrete design elements (such as classes and subsystems).

Map the software to the hardware Architecturally significant design elements are mapped to the target deployment environment.

Validate the architecture Development work should be performed to build a new version of the system demonstrating that the architecture is viable. During the

[1] see e.g. `http://www.controlchaos.com/`

early stages of a project the primary goal of this phase is to generate prototypes which show the suitability of the architecture and provide a stable foundation for the remaining development work.

As foreseen in the TripCom description of work, we are constraint to develop the Triple Space architecture within a single iteration of the phases introduced above. Nevertheless the drawbacks inherently induced by this sequential approach are compensated by a careful elaboration of the initial two phases in the process. The architectural mechanisms available as well as the overall objectives of the architecture have already been discussed within the first 18 months of the project, leading to a broad understanding of the design elements required to realize a scalable Triple Space infrastructure [6]. The work reported in this deliverable is based on the reference architecture from [6].

2.1.1 Architectural Views

An architecture can be represented from a variety of viewpoints, which can be combined towards an overall view upon the system. Each architectural view addresses some specific set of concerns, specific to stakeholders in the development process: users, designers, developers, managers etc. We use a model similar to [3], differentiating among the following views:

Use-case view This view focuses on the system from a user perspective. Consequently it contains a use case model and describes the functionality of the system, its users and the way they interact with the system in terms of interfaces. We use UML Use Case diagrams to create this view.

Logical view This view describes the system in terms of units of implementation packages and the relationships between them. Logical views serve as an overview of the general architectural principles followed to design the system; this is best achieved using a simple diagram using basic chapes such as boxes and arrows, identifiying key components and their relationships among each other.

Data view This view describes the data model of the system. We use this view to identify and design data that flows between individual components. We describe the data flow between components in the form of a custom component-component matrix as there is no established method of describing data flow for applications based on a tuplespace based system.

Process view This view is important for analyzing the run-time system parameters such as reliability or performance. It describes the system in terms of set of elements with run-time behavior and their interaction. Depending on the technology used such elements can be processes, threads, DLLs, but also data stores and connectors such as queues. We use standard UML Activity and Sequence diagrams to provide this view.

2.1.2 The Triple Space Architecture

Figure 2.2 from [6] shows the logical view (cf. Section 2.1.1) on a single Triple Space kernel with both, kernel-internal components and kernel-external clients and services

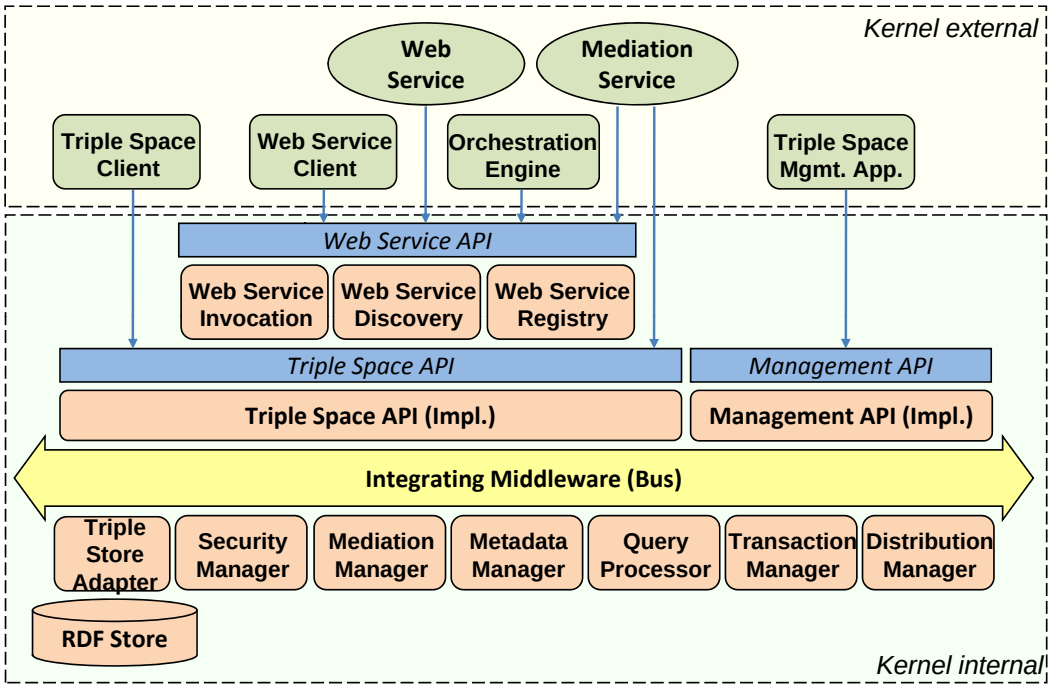

Figure 2.2: Triple Space Architecture – Logical View [6]

that may be connected to it. From a birds-eye view on the architecture one can see two different areas. The upper area shows kernel-external entities that may connect to the kernel. The area below shows the components of a kernel itself, i.e. APIs and components implementing these APIs. The components that form a Triple Space kernel communicate over a kernel-internal bus system which is implemented using a space-based middleware system. The integrating middleware allows all components that are drawn directly above or below to communicate with each other in a bus-like manner. The bus itself is depicted as a yellow arrow in the lower part of the figure.

APIs (blue boxes) describe logical groups of functionality through visible interfaces, hiding the actual implementation and are only accessible in a top-down manner; i.e. it is not allowed that a lower level entity accesses an entity above it.

Triple Space clients (green boxes) and *services* (green circles) are kernel external and interact with the Triple Space using the Triple Space API (defined by WP3 in [7]), the Web Service API (defined in WP4), and the Management API (defined by WP3 and WP5).

Mediation services provide kernel-external transformation functionality used by the mediation manager. An

Orchestration engine can be used to build higher-level functionality based on single Web services through service composition e.g. using a BPEL [2] engine.

Components (orange boxes with round corners) directly below an API implement the API's functionality. Components connected to the integrating middleware can only interact with other components over the bus - they are not allowed to interact directly. Kernel-internal components above the Triple Space API (e.g. the Web service invocation component) communicate using the Triple Space API in the same way as kernel-external entities do.

2.2 Coordination Patterns

When considering the component interaction in a Triple Space kernel, it is important to identify the coordination patterns to be supported by the kernel as each pattern may need to be implemented by a different set of interactions between the internal components. These coordination patterns can be specified on the basis of the public Triple Space API.

The Triple Space API has been revised since the publication of a first version in [7] as a result of further discussions, insights won through implementation and the need to provide different configurations of the API to account for scalability and expressivity trade-offs. The revised API is given in this section. It allows for three levels of expressivity (which reflect the three configurations of Triple Space), i.e. the number of coordination patterns supported increase as one moves from the Core API level to Extended (e.g. adds publish/subscribe) and Further Extended APIs (e.g. adds transactionality). While the implementation of API operation support differs between components (e.g. for the Transaction Manager, it only makes sense once the Further Extended API is supported), for the first implementation Core API support was decided as being the minimal requirement. Hence, when considering the coordination patterns to support in terms of component interactions (see section 3.4) we can base our implementation in terms of the basic operations of the Core API: the emission of single triples (out) and the non-destructive retrieval of a single match from either a named space or from the Triple Space (rd with or without space URL, respectively).

The following tables explain the API operations using abstract classes. In API groundings, these classes are mapped to actual classes in a chosen API. Remote groundings such as SOAP define their own operations for request and response actions, and the API operations are contained within the communication content (e.g. SOAP message) typically as string values of an "operation" parameter. Details of the API groundings will be specified in a WP3 report.

The definitions of the used abstract classes are:

- Triple – atomic data unit equivalent to a RDF statement
- Set¡Triple¿ - a set of Triple objects, which forms as a whole a single RDF graph
- Template – generic query
- SimpleTemplate – a triple pattern (triple which can contain variables); the simplest query possible
- Space – a reference to a named triplespace
- Time – a time measurement to indicate the duration for which a kernel shall wait for the resolution of the query before unblocking the requesting process
- Callback - represents a callback object in whichever manner this may be supported by the chosen grounding

Core API

Operation	Returns	Description

out(Triple t, URL space)	void	Atomically writes a single triple into the space. The operation makes no guarantee if and when the triple will be available in the space (unordered semantics). The client is immediately free to perform further activities. The client has to provide a resolvable URL which identifies a space.
rd(SingleTemplate t, URL space, Time timeout)	Set<Triple> s	Returns one match of the given template which is a single triple pattern. The match may be a set of triples, e.g. Concise Bounded Description. The operation makes no guarantee as to when the match would be returned to the client. A timeout is provided to give a temporal bound for returning a match. If no match has been found by the timeout period, an empty set is returned. This does not make any statement regarding the existence of a match in the space. No timeout can be specified by providing a null value to the timeout parameter.
rd(SingleTemplate t, Time timeout)	Set<Triple> s	As the rd operation above but no space URL is specified. The system is free to select a match from anywhere in Triple Space where the client has read permissions.

Extended API

Operation	Returns	Description
out(Set<Triple> t, URL space)	void	Atomically writes a set of triples into the space. The operation makes no guarantee if and when the triples will be available in the space (unordered semantics). The client is immediately free to perform further activities. The client has to provide a resolvable URL which identifies a space.
rd(Template t, URL space, Time timeout)	Set<Triple> s	Returns one match of the given template. The match may be a set of triples, e.g. Concise Bounded Description. The operation makes no guarantee as to when the match would be returned to the client.A timeout is provided to give a temporal bound for returning a match. If no match has been found by the timeout period, an empty set is returned. This does not make any statement regarding the existence of a match in the space. No timeout can be specified by providing a null value to the timeout parameter. More expressive templates can be supported by Triple Space implementations than the single triple pattern of the Core API, e.g. SPARQL Queries. The actual set of triples returned will be determined by the definition of the matching rules.
rd(Template t, Time timeout)	Set<Triple> s	As the rd operation above but no space URL is specified. The system is free to select a match from anywhere in Triple Space where the client has read permissions.

rdmultiple(Template t, URL space, Time timeout)	Set<Set-<Triple>>s	As the rd operation above but returns multiple matches of the given template. Each match may be a set of triples, e.g. Concise Bounded Description. There is no completeness guarantee, i.e. the answer given to the client may not contain all matches within the space.
subscribe(Template t, Callback c, URL space)	URI subscription	A subscription is established by providing a template and a callback. When a set of triples matching the template is atomically outed into the specified space, the callback is sent that matched set of triples. The operation returns an URI identifying the subscription.
unsubscribe(URI subscription)	void	This cancels the active subscription with the given URI if it exists.

Further Extended API

Operation	Returns	Description
in(Template t, URL space, Time timeout)	Set<Triple>	As rd, but deletes eventually within the given space the matched triples as determined by the matching rules. It is possible that Triple Space, with the principle of persistent publication, does not need any destructive read functionality.
inmultiple(Template t, URL space, Time timeout)	Set<Set-<Triple>>	As rdmultiple, but deletes eventually multiple matched triples as determined by the matching rules.
createTransaction(String type)	URI transactionID	Type can be “local” or “shared”. The idea is to support both transactions which are local to a client (URI is only known to the client) and are shareable with others (for distributed transactions). Identifiers of shared transactions would be made available to other clients, who can then ’get’ the shared transaction to participate in it (see getTransaction). Transactions may be supported optimistically or pessimistically, the latter would in the case of Triple Space potentially make less guarantees (weak ACIDity).
getTransaction(URI transactionID)	Boolean result	Used to join in a shared transaction with other clients. If true is returned, the client now shares in this transaction once it is begun until it is committed or rolled back.
beginTransaction(URI transactionID)	Boolean result	Begins the transaction. All subsequent interactions by all agents sharing this transaction are handled transactionally, i.e. ’all or nothing’.
commitTransaction(URI transactionID)	Boolean result	commits all interactions made within this transaction in the space.
rollbackTransaction(URI transactionID)	Boolean result	rolls back all interactions made within this transaction in the space.

2.2.1 Management API

Furthermore, a separate API has been developed for operations carried out by kernel or space administrators, and hence do not belong in the public Triple Space API for normal clients. The Management API was derived from the requirements of components in Triple Space where changes had to be effected by administrators outside of the kernel. Currently, these operations can be divided into three types:

- Metadata administration: adding and deleting metadata about the kernel, spaces and data. (used by Metadata Manager)
- Space administration: adding and deleting spaces on the kernel. (used by TS Adapter, triggers update in Metadata Manager)
- Security administration: adding and deleting access policies for clients accessing data on the kernel. (used by Security Manager)

Operation	Returns	Description
children(URL space)	Set<URL>	Provides a list of identifiers for those spaces which are subspaces of the given space.
create(String path, URL space)	URL	The client provides a path and a space and the operation returns a new URL which identifies the newly created subspace. Its use will be controlled by the access control policies; as a rule only the space administrator can create new subspaces in their space and only the kernel administrator can create new subspaces of the root space.
destroy(URL space)	void	This deletes eventually the space identified by the given URL. This also eventually deletes all child spaces of this space. This functionality may be too destructive and breaks the principle of persistent publication in Triple Space, hence it may not part of the default Management API but offered only on certain kernels.
addMetadata(URL space, Graph g)	void	This operation adds the triples provided in the RDF Graph g to the metadata for the space with the given URL. For this operation to successfully execute, the space given must reside on the kernel being contacted.
removeMetadata(URL space, Graph g)	void	This operation removes the triples provided in the RDF Graph g from the metadata for the space with the given URL. Only those triples in the graph which match triples in the stored metadata are removed, the other triples in the graph will be ignored. Blank nodes will be treated as wildcards, removing all triples which match on all the constants in the graph's triple. For this operation to successfully execute, the space given must reside on the kernel being contacted.

Table 2.4: Management API operations for space management

Operation	Returns	Description
m-out(Graph g, Space s)	boolean	Adds the access control policies in graph g to the access control policies for space s. If space s is null, the access control policies apply to the kernel root space to which this operation was passed. The operation returns true once the access control policies have been added at the kernel. False can indicate that the policies were invalid, e.g. not conform to the policy language supported by the kernel or that the space s is not managed by that kernel.
m-rd(Query q, Space s)	RDF Graph	Applies the query q to the access policies applying to space s and returns the result as a RDF graph. The query can be any supported at the kernel (as defined in WP3). We can expect SPARQL to be supported. As policies will be self-contained RDF graphs, the answer is always made up of complete policies. In other words, if part of a policy matches the query, the entire policy is returned. If space s is null, the operation applies to the kernel access policies.
m-in(Query q, Space s)	RDF Graph	As m-rd but deletes the policies returned in the RDF Graph answer to the query.
m-update(Graph g-in, Graph g-out, Space s)	void	Changes the access control policies for space s in that all triples matching triples in Graph g-in are removed, and all triples in Graph g-out are added. Graph g-in may use blank nodes as wildcards, removing all triples which match the constants in the graph's triple. If space s is null, this applies to the kernel access policies.

Table 2.5: Management API operations for security

2.3 Component Descriptions

The Triple Space architecture foresees various components which communicate internally in a kernel. Prior to the specification of their integration, to realise the coordination patterns defined by the Triple Space API, we describe the components in terms of individual functionalities and dependencies on other components. To do this:

- We follow and extend the approach from [6] i.e. list all components and describe their implementation;
- This forms the documentation and the reference document for the prototypical implementation.

2.3.1 Triple Space API

The Triple Space API component realises the implementation of the Triple Space API operations by checking and packaging their content appropriately and inserting the Entry representing the operation into the system bus. It also monitors for responses, taking them out of the system bus, unpackaging the contents and returning an answer to the client. Finally, it acts as a garbage collector, i.e. "hanging" operations in the system bus may be removed and the client notified accordingly.

The API grounding interface provides the mapping from an external communication structure (protocol and data) to the internal implementation structure for each

operation. There can be multiple interfaces supported by an API component on one kernel. The interface can also support additional functionality such as returning error messages to clients communicating incorrectly with the interface or providing an interface description to clients requesting it. We will support Java (simple types), Java (openRDF) and SOAP.

Use Cases

- Actors: a client and the system bus
- Use cases: each API operation
- A client passes an API operation to the kernel. Depending on the operation, the client may then continue to function, or may block until a response is received. The kernel attempts to perform the operation. Eventually, either a response is sent or an error message (NB. "out" does not return any response to the client).
- The use cases are related through the functionality of the individual operations.

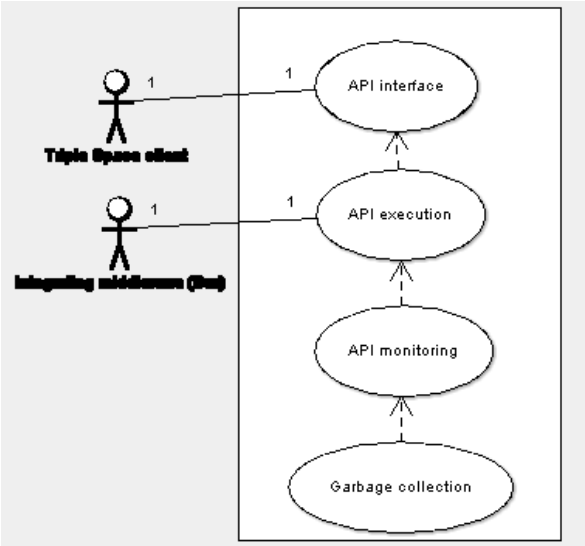

Figure 2.3: Use cases for the TS API Implementation

2.3.2 Management API

The Management API is in concept and function very similar to the TS API component, except that it supports operations for kernel and space administrators.

Use Cases

- Actors: an administrator and the kernel bus (XVSM)
- Use cases: each Management API operation

Use case: API grounding interface	
Description	Makes client access to the Triple Space kernel possible over a certain protocol and at a certain address. A communication structure (data and protocol) is defined for the correct expression of Triple Space requests.
Pre-conditions	A client sends data to the interface using the protocol it supports and the address it makes itself available over.
Post-conditions	An API request is passed to the API implementation or an error message is returned to the client.
Normal flow	1. Check if access is appropriate (e.g. virus scan, identification of Denial of Service attacks) 2. Check the (syntactic) validity of the data passed to the interface 3. Map data into an API request which is passed to the API implementation
Error situations	The access is inappropriate. The data sent is not valid.
Error flow	1. The component may return an error to the client. 2. The component may take protective measures, e.g. banning access from that client to the kernel.

Table 2.6: Use case: API interface

Use case: API operation execution	
Description	Realises the functionality for the API operations
Pre-conditions	An API operation is passed into the API implementation
Post-conditions	A tuple (representing the API operation) is inserted into the system bus
Normal flow	1. Check and package the operation content into a message (tuple) 2. Insert this message into the system bus
Error situations	The component could not package the operation content into a message due to some error in the content. The component could not insert the message into the system bus.
Error flow	1. The component may try to resolve the error itself, e.g. to fix the content or wait and retry to access XVSM. 2. An error message may be sent back to the client notifying it of erroneous content or system unavailability.

Table 2.7: Use case: API operation execution

Use case: API operation monitoring	
Description	Monitors the resolution of active API requests
Pre-conditions	An API operation is inserted into the system bus
Post-conditions	The operation has successfully completed, or the operation has failed
Normal flow	1. Create a notification on the system bus for a resolution of the operation 2. Wait for a callback 3. The callback contains either a success or failure notification 4. Return operation response or failure message to client
Error situations	A notification could not be created. The notification never ends in a callback (NB. definition of "never" determined by the component).
Error flow	1. The component should wait a period of time and try again (a notification is necessary for the resolution of an operation.) 2. A "hanging" notification may be picked up by the garbage collector.

Table 2.8: Use case: API operation monitoring

- A client passes a Management API operation to the kernel. The client waits for a response. The kernel attempts to perform the operation. Eventually, either a response is sent or an error message.
- The use cases are related through the functionality of the individual operations.

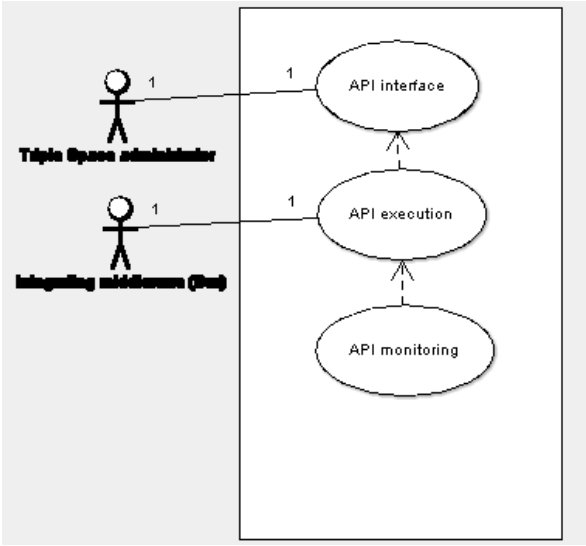

Figure 2.4: Use cases for the Management API

2.3.3 Triple Store Adapter

Description

Triple Store Adapter (TSA) is a flexible and configurable component to abstract the underlying storage infrastructure and provide a coordination interface to other components. The main component functionality is oriented toward the efficient management of large amounts of data and more specifically the support of:

- Persistent storage of RDF based data and the associated metadata until explicitly removed by a user or system request.
- Evaluation of RDF queries against the stored semantic data.
- Optional support of efficient reasoning capabilities over the stored information (TRREE implementation, see D2.3 [5])
- Optional support of distribution of the information (YARS implementation, see D2.3 [5])

Use Cases

Three use cases related to the TSA are identified. The first two use cases define how RDF based data is written and removed from the storage, while the third specify the execution of RDF queries.

Use case: API operation garbage collection	
Description	Optimises kernel handling of API requests by using a rule of thumb to remove "hanging" operations which are using up kernel resources.
Pre-conditions	An API operation has been active for a significantly long period of time without resolution.
Post-conditions	The API operation is removed from the system bus.
Normal flow	1. Check for notifications in the API monitor which have not yet been resolved after a significant amount of time (NB. the definition of significant would be made within the component) 2. Check for inactivity of the client which made the API request on that kernel 3. Delete the notification from the API monitor. 4. Check for resolved operations in the system bus for which the API monitor was not notified (e.g. because of step 3) 5. Remove the API operations from the system bus (NB. potentially cache them and repeat it at a later time, e.g. when the client becomes active again)
Error situations	None. The removal of the API operation from the system bus should trigger other components or kernels to cease handling it.
Error flow	None.

Table 2.9: Use case: garbage collection

Use case: Management API interface	
Description	Makes client access to the Triple Space kernel possible over a certain protocol and at a certain address. A communication structure is defined for the correct expression of Triple Space management requests.
Pre-conditions	A client sends data to the interface using the protocol it supports and the address it makes itself available over. The data may be encrypted.
Post-conditions	A management API request is passed to the Management API implementation or an error message is returned to the client.
Normal flow	1. If necessary, decrypt the message using the decryption key associated to this administrator. 2. Check if message content is appropriate (e.g. virus scan, identification of Denial of Service attacks) 3. Check the (syntactic) validity of the data passed to the interface 4. Map data into a management API request which is passed to the Management API implementation
Error situations	The message can not be decrypted The access is inappropriate. The data sent is not valid.
Error flow	1. The component may return an error to the administrator. 2. The component may take protective measures, e.g. banning access from that administrator to the kernel.

Table 2.10: Use case: Management API interface

Use case: Management API operation execution	
Description	Realises the functionality for the Management API operations
Pre-conditions	A Management API operation is passed into the Management API implementation
Post-conditions	A tuple (representing the Management API operation) is inserted into the system bus
Normal flow	1. Check and package the operation content into a message (tuple) 2. Insert this message into the system bus
Error situations	The component could not package the operation content into a message due to some error in the content. The component could not insert the message into the system bus.
Error flow	1. The component may try to resolve the error itself, e.g. to fix the content or wait and retry to access the bus. 2. An error message may be sent back to the administrator notifying it of erroneous content or system unavailability.

Table 2.11: Use case: Management API operation execution

Use case: Management API operation monitoring	
Description	Monitors the resolution of active Management API requests
Pre-conditions	A Management API operation is inserted into the system bus
Post-conditions	The operation has successfully completed, or the operation has failed
Normal flow	1. Create a notification on the system bus for a resolution of the operation 2. Wait for a callback 3. The callback contains either a success or failure notification 4. Return operation response or failure message to administrator
Error situations	A notification could not be created. The notification never ends in a callback (NB. definition of "never" determined by the component).
Error flow	1. The component should wait a period of time and try again (a notification is necessary for the resolution of an operation.) 2. The component will consider the operation as failed and cancel it, informing the administrator. The notification is removed as well as the active operation tuple in the system bus.

Table 2.12: Use case: Management API operation monitoring

Use case: Persist semantic data	
Description	Specific data must be permanently persisted
Pre-conditions	Component requests semantic data to be stored
Post-conditions	Semantic data is permanently stored
Normal flow	1. TSA is notified for the existence of new data to be persisted 2. TSA destructively reads the new data to be persisted 3. TSA stores the triple to the underlying storage
Error situations	1. The persisted data does not conform the RDF data model 2. Internal storage exception is triggered.
Error flow	1. The data is written back to the space and is marked as invalid. 2. The data is written back to the space and the exception is attached to it. from that client to the kernel.

Table 2.13: Use case: Persist semantic data

Use case: Remove semantic triples	
Description	Specific data must be permanently removed from the storage
Pre-conditions	Component requests to remove semantic data
Post-conditions	The semantic data is removed from the storage
Normal flow	1. TSA is notified for the existence of new remove pattern request 2. TSA destructively reads the remove pattern request 3. TSA writes back the pattern and the number of deleted triples
Error situations	1. The values in the pattern do not conforme the RDF data model Internal storage exception is triggered.
Error flow	1. The remove pattern is written back to the space and is marked as invalid. 2. The data is written back to the space and the exception is attached to it. from that client to the kernel.

Table 2.14: Use case: Remove semantic triples

Use case: Evaluates RDF query	
Description	RDF query to be evaluated against the stored semantic data
Pre-conditions	Component need to retrieve semantic data
Post-conditions	Query result is written in the space
Normal flow	1. TSA is notified for the existence of new RDF query request 2. TSA destructively reads the RDF query request 3. TSA writes back the query result set
Error situations	1. The RDF query has malformed format 2. Internal storage exception is triggered.
Error flow	1. The remove pattern is written back to the space and is marked as invalid. 2. The data is written back to the space and the exception is attached to it. from that client to the kernel.

Table 2.15: Use case: Evaluation of RDF query

2.3.4 Query Pre-Processor

Query PreProcessor is responsible for processing the query that is initially made by user through TS API to access, manipulate or publish any triples. The Triple Space Computing is a distributed system that consists of different interconnected kernels. Each of the kernels contains RDF repositories and contains a local query processor (based on SPARQL). The term Query PreProcessor should not be confused with the local Query Processor of a kernel. The Query Pre-processor takes care of reasoning, optimizing the query before its execution, checks for any possible inconsistencies, as well as takes care of horizontal growth of data in cooperation with Distribution Manager. It enables the processing of queries that scales not only vertically (i.e. increasing local resources within one TripCom Kernel), but also horizontally (i.e. increasing number of interconnected kernels). The local Query Processor in a kernel is a simple SPARQL query processor that received the local queries from Query PreProcessor and executes it at local RDF data stores within the kernel.

The Query PreProcessor consists of two major sub-components namely Query Optimizer and Inconsistency Reasoner. The Query Optimizer during the processing of a user's query communicates with Distribution Manager and the Triple Storage Adapter. The sole purpose of the Query Optimizer, sitting on top of the SPARQL query processor, is the expansion of the "locally working" SPARQL query processor of each kernel to an adaptive distributed operation mode.

The purpose of Inconsistency Reasoner is to take the results obtained from local Query Processor at TripCom data access layer (based on the query provided by Query Optimizer) and to process it to detect and avoid any inconsistencies in the results as a first step. Secondly, it tries to deduce any further possible meaningful results from the incomplete data. It ensures that there is no inconsistency in the results obtained at the local Query Processor and makes the data ready to be sent to Distribution Manager.

The Query PreProcessor handles following:

- Query from Distribution Manager that was submitted by clients through TS API
- Interaction with Distribution Manager for iterative answer reconstruction
- Sending query to the local Query Processor at Triple Store Adapter
- Accessing the local RDF schema for Inconsistency reasoner
- Marking inconsistency check over the results obtained from local Query Processor

Use Cases

- Actors: a query engine and the kernel bus
- Use cases: stages of execution of a query
- The Query Pre-Processor will take the query from Distribution Manager and Query Optimizer will perform cost estimation as well as necessary query rewriting. The local query generated by the Query Optimizer will be sent to the local Query Processor of the Triple Store Adapter. The Inconsistency reasoner will

check for any inconsistencies in the results obtained from the local query processor and pass it to the Query Optimizer to send it to the Distribution Manager to perform iterative answer reconstruction for the results obtained from other kernels.

- The use cases are related through the workflow of the query pre-processing

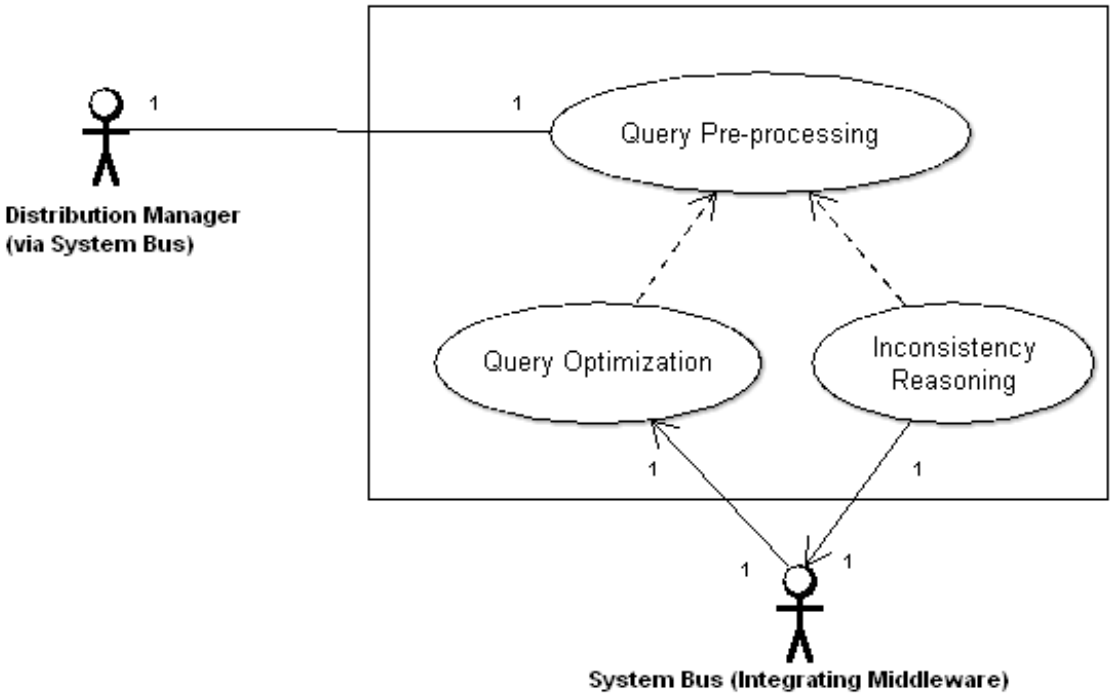

Figure 2.5: Use cases for the Query Pre-Processor

2.3.5 Distribution Manager

The Distribution Manager takes care of incoming operations (from clients or other kernels) and passes them on to the system bus of the appropriate kernel, whether it is the local kernel or a remote kernel.

To detail this a bit more, we can identify four cases which the Distribution Manager of a kernel could handle:

- 1. An operation from a client on that kernel which can be resolved on that kernel.
- 2. An operation from a client on that kernel which must be resolved on another kernel.
- 3. A response from another kernel to an operation which originated on this kernel.
- 4. A response from another kernel to an operation which originated on some other kernel.

Use case: Query Optimizer - Cost Estimation	
Description	Retrieves an active query and omptimizes it so that it is ready to be handed off to adequate query processors , i.e. the local TS Adapter and/or remote kernels.
Pre-conditions	A query has been inserted into the system bus.
Post-conditions	The query exists in a form which can be accepted by a selection of query processors, i.e.the local TS Adapter and/or remote kernels.
Normal flow	1. A notification is triggered that a query is present in the system bus 2. The query is removed from the system bus 3. A tuple is inserted in the system bus stating that the query operation is being processed in the Query Pre-Processor 4. Several optimization techniques (rewritting, cost estimation, join order selection, decomposition) are applied to the query. 5. Appropriate kernels are chosen based on the results of step 4 6. For each kernel selected in step 5 the query is mapped to a form required by the query engines embodied in its TS Adapter
Error situations The query engine can not be determined.	The query type can not be determined (e.g. query syntax is wrong or uses a function not recognised by the component such as a non-standard extension) The format required for the query engine can not be generated.
Error flow	1. The component may return an error to the client. 2. The component may try to take corrective measures e.g. self-correction of query syntax. However this has the danger of interpreting the query differently from what the client intended.

Table 2.16: Use case: Query processing

Use case: Query Answering	
Description	Resolves a query to a result set
Pre-conditions	A query is ready to be passed to the local TS Adapter and/or chosen remote kernels.
Post-conditions	A result set is returned from the local TS Adapter and/or chosen remote kernels.
Normal flow	1. The query is passed to the local TS Adapter and/or chosen remote kernels. 2. A result is received back from the local TS Adapter and/or chosen remote kernels.
Error situations	The local TS Adpater/a remote kernel returns an error. The local TS Adpater/a remote kernel does not respond.
Error flow	1. The component may return an error to the client. 2. The component may try to take corrective measures e.g. tries to resolve the query with another kernel. Care should be taken to do this only when the same result could be expected.

Table 2.17: Use case: Query answering

Use case: Inconsistency Detection	
Description	Checks for the results obtained from local query processor and checks for any inconsistency that may arise.
Pre-conditions	A result set has been received from the query engine.
Post-conditions	The result set exists without any inconsistency with respect to the local RDF schemas and is in a form which can be returned to the client.
Normal flow	1. The result set is analyzed for detecting any possible inconsistency. 2. The query will be analyzed against the ontology (in this case RDF schema). 3. Answer of the query will be ranked as either over-determined, accepted, rejected or undetermined. 4. If the result is obtained as "accepted", no inconsistency reasoning is performed. 5. If result is obtained as over-determined, inconsistent reasoning is performed. Otherwise, result is rejected and not processed further.
Error situations	Post-processing can not produce a "valid" result for the client.
Error flow	1. The component may return an error to the client. 2. The component may try to take corrective measures e.g. some heuristic to generate valid results. The danger is that the results will be incorrectly delivered to the client.

Table 2.18: Use case: Inconsistency Detection

Use case: Inconsistency Reasoning	
Description	Uses selection functions to determine which consistent sub-sets of an inconsistent ontology should be considered in the reasoning process.
Pre-conditions	The result from the local query processor is evaluated as over-determined.
Post-conditions	A subset of the result is returned that contains no inconsistency, and is still in a form that can be sent to the user.
Normal flow	1. A monotonic selection function is applied that checks for a subset until it finds a consistent one. 2. Inconsistency Reasoner uses linear extension to find out minimal consistent subset. 3. Answer is check that if it is locally sound and complete to a consistent subset of ontology. 4. This process is repeated until it finds out a consistent subset of ontology and a valid result against it.
Error situations	Post-processing can not produce a consistent subset of ontology.
Error flow	1. The component may return an error that it cannot find out any consistent subset of ontology and the ontology may be flawed.

Table 2.19: Use case: Inconsistency Reasoning

For case (1) we can postulate that such operations, when placed in the system bus, can be picked up from the relevant kernel component and are not taken by the Distribution Manager, as the kernel can know itself which spaces it hosts. However, in any other case, the case (2) is relevant and it is the Distribution Manager which must handle the operation. If other kernels are contacted, then a response should also be received from them, and as a response to an operation which originated on the kernel, we have case (3). Finally, dependant on the deployment architecture chosen by Triple Space, it may be that we have case (4) where the kernel acts purely as router between two other kernels. This can be the case, for example, with hops in a P2P network, as the Distribution Manager will provide the local P2P implementation through the network interface subcomponent. However, this case will only occur in communication to and from hash tables which are distributed across kernels in the Triple Space and which uses the underlying P2P infrastructure; communication between kernels is expected to be point-to-point using the Web infrastructure (DNS and TCP/IP).

Hence we focus mainly on cases (2), (3) and (4). Case (1) may not be necessary, and if so it is simply a form of case (2) where the kernel that is identified for handling the operation will be the kernel on which that component runs.

To achieve this, the Distribution Manager needs functional support for two types of mapping: a space-to-kernel mapping and a template-to-space mapping. In the current implementation, the former can be resolved using the DNS infrastructure of the Web. The latter will use a Distributed Hash Table based approach.

Furthermore, it requires a monitor for waiting on and collecting responses from kernels. Finally, it will need an interface to the deployment (physical) architecture of the Triple Space, which we expect will also contain necessary network functionality such as secure, reliable messaging, packing operations and responses into appropriate data formats and the physical routing of the messages between kernels.

Here, let us differentiate between two forms of inter-kernel communication. Firstly, template-to-space mapping is expected to use hash tables and the underlying P2P infrastructure to distribute key-value pairs among hash tables stored on Distribution Managers on each kernel. Each hash table covers a certain segment of the entire key space. Here, the P2P network is used to send messages to add entries in tables, retrieve a value using a certain key and possibly delete entries in tables. Secondly, operation passing will use the Web infrastructure to communication directly to target kernels using TCP/IP and probably HTTP.

Use Cases

- Actors: the Triple Space network of kernels and the local kernel bus (XVSM)
- Use cases: cases (2), (3) and (4) above.
- An operation is passed into the kernel. The Distribution Manager determines to which kernels it should be passed. If no space URL is given, first the template-to-space mapping is used. If a space is given, or a set of spaces determined from the template-to-space mapping, then the space-to-kernel mapping is used to find the kernels to be called. The network interface handles contacting the kernels. The monitor tracks responses until the operation is completed and returns the completed response to the client over the system bus, If the operation times out, those responses received until the time out can be returned. In the meantime,

the network interface may be forwarding nessages from other kernels to other kernels.

- The use cases represent the different concurrent activities that take place in the component.

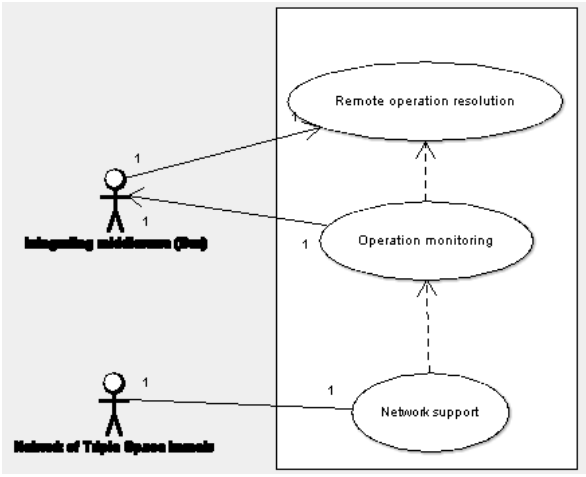

Figure 2.6: Use cases for the Distribution Manager

Use case: Remote kernel operations	
Description	Determines the kernel(s) to be called to resolve an operation received at this kernel.
Pre-conditions	An operation published in the system bus which either has no space URL or gives a space URL which is not hosted at this kernel.
Post-conditions	The operation is forwarded to the relevant kernel(s) and the monitor activated.
Normal flow	1. If no space URL is given, use the template-to-space mapping functionality to determine a set of relevant spaces 2. For each space URL, use the space-to-kernel mapping functionality to determine the kernels to call. 3. Pass operation with a list of kernels to the network interface and activate the monitor for that operation.
Error situations	The mapping functionality is not available. No relevant space or kernel can be found.
Error flow	1. The component may return an error to the client, e.g. if a space is given that does not exist. 2. The component may attempt an alternative, e.g. if the mapping functionality is an external service and it is not responding.

Table 2.20: Use case: Remote kernel operations

2.3.6 Metadata Manager

The Metadata Manager provides support to the operation of the Triple Space kernel by generating and storing metadata according to the Triple Space ontology [4].

This metadata describes semantically details of the spaces and data on kernels as well as metadata for those spaces and data such as relationships between spaces, creation details for data, access logs etc. It can be used to complement queries on a space or to improve the data look-up in Triple Space.

Use case: Operation monitoring	
Description	For operations originating in the kernel which have been passed to other kernels, wait on those kernels' response, collect the responses and handle error conditions.
Pre-conditions	An operation has been passed off to another kernel through the network interface.
Post-conditions	The response to the operation is prepared and returned to the client.
Normal flow	1. A monitor entry is created for an operation with a record of which kernels were called 2. For each response received for that operation, record that in the monitor entry. 3. For long-lasting operations, if no response is received from a kernel after a certain period of time, the monitor can choose to retry sending a message to that kernel. (May cause unnecessary overhead. Messaging should be reliable and it should be possible to generate error messages if a kernel is not available) 4. For error responses, the monitor can determine to retry a kernel, or select other kernels to call. 5. Once a final state has been reached, e.g. timeout of the operation or all kernels have responded, the results can be returned to the system bus and the monitor entry is removed.
Error situations	Kernels may return error messages such as client not having access rights or may not be available on the network.
Error flow	1. The monitor may accept the error, given that other kernels have provided npn-error situations or no other kernels come into question. 2. The monitor may choose to retry kernels or call other kernels which would be relevant for handling the operation.

Table 2.21: Use case: Operation monitoring

Use case: Network support	
Description	Operations from within the kernel are passed into the Triple Space network to be routed to remote kernels. Responses from remote kernels are also to be forwarded or placed within the kernel. Note that the network can be the P2P implementation (for hash table routing) or the Web infrastructure (for operation passing).
Pre-conditions	A message is to be sent over the Triple Space network or is received from the network.
Post-conditions	The message has been dealt with, either by being taken by the kernel or being routed elsewhere.
Normal flow	1. A message is ready to be sent on the network or ready to be received from the network. 2. Outgoing messages are placed into the network with the destination address of the kernel to be called. 3. Incoming messages are examined for their destination address. 4. If the destination address is the kernel, they are taken from the network and placed in the system bus of the kernel. 5. If the destination address is another kernel, the component acts as a forwarder.
Error situations	The network at the kernel may be down. Incoming messages may not be interpretable (e.g. corrupted) Incoming messages contain a destination address that can not be resolved.
Error flow	1. If no network is available, the network interface may be able to queue ougoing messages until the network becomes available, to a reasonable extent. 2. If incoming messages can not be handled, the network interface may send error messages to the kernels informing them of the problem.

Table 2.22: Use case: Network support

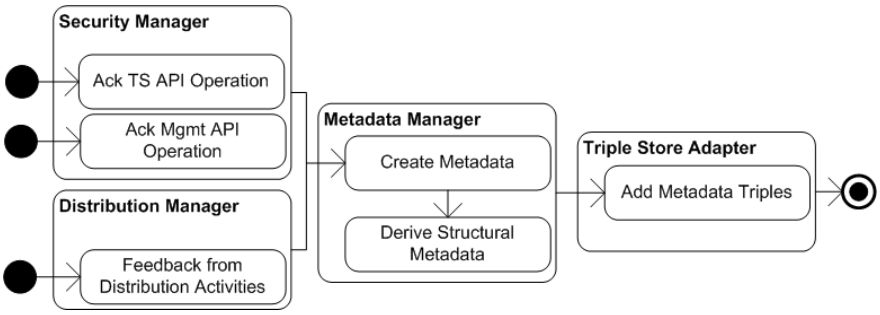

Figure 2.7: Use case 1: creation of metadata

Description

The Metadata Manager is responsible for the creation and processing of metadata. Metadata in TripCom refers to the administrative data about the space deployment, structure and the application data (user data) published therein. This includes the modelling of spaces, data and kernels, and their characteristics, data access logs and distribution within the overall Triple Space system.

The Metadata Manager is involved in three procedures:

1. creation: the Metadata Manager creates metadata based on requests and information delivered by other kernel components. In particular, the Metadata Manager reacts to structural changes reported by kernel components, and to the publication and retrieval of user data.
2. processing: the Metadata Manager might derive additional data from the collected one, or resolve management rules that again result in additional knowledge about the behavioral and structural state of the Triple Space.
3. delivery: the Metadata Manager provides access to the collected and derived metadata for other components. The primary demands will come from the Distribution Manager, which needs routing information in order to forward published triples, and/or request for distributed retrievals.

In order to better describe the functionality of the Metadata Manager we present in the Use Cases section some representative examples for the tasks one (1) and three (3). The second one, as being a purely internal functionality to the component, is seen to be less indicative with respect to the component's functionality and thus left aside.

Use Cases

In this part we explain the functionality of the Metadata Manager by help of two use cases (Figures 2.7 and 2.8).

Actors: All TS kernel components that either deliver information about the structure of a space, or about the data published and retrieved.

Use Case 1: The Metadata Manager creates content and structural metadata based on requests and information delivered by the Security Manager (API calls) and the Distribution Manager (routing information) component (Figure 2.7).

Preconditions: The user data is successfully written to the kernel by use of the TS API and fulfilling all security requirements, which directly triggers the

annotation of user data, or indirectly the management of structural information about space hierarchies and distributed retrieval.

Activities: The Metadata Manager consumes the triples and additional data about the triples (publisher...), or management operations that influence the space structures of Triple Space - these actions are triggered by the Security Manager. Moreover, this component reacts to distribution and routing feedback from the Distribution Manager. The so provisioned metadata is serialized to RDF according to the TS Ontology [4] and fed back to the Integration Middleware in order to be persistently stored via the Triple Store Adapter. In particular the structural metadata is then further processed to derive semantic routing tables.

Errors: The Metadata Manager has no influence on the functionality of Triple Space, but rather provides a support component that enables improved handling of non-functional aspects. There are no safety threats expected from this component.

Use case 3: The Metadata Manager provides access to the collected and derived metadata for other components. There are two ways that other components can gain access to the knowledge of the Metadata Manager. First, the Metadata Manager provides routing information for the Distribution Manager – this is the use case we focus on here (Figure 2.8). Second, the Metadata Manager uses its knowledge to inspect the structure and behavior of the Triple Space and trigger adaptations if required. This is primarily part of the self-organization taks T2.8 and hence left aside for now.

Preconditions: There must be metadata stored with the Metadata Manager.

Activities: Based on the data provided, as in use case 1, this component derives further facts about the content and the structure of the space. This information is for example used to establish access logs about user data or semantic routing tables for the Distribution Manager.

Errors: The structural metadata derived might be false, which can be seen as not being an error, but simply a sub-optimal support action. In that sense there are not critical tasks involved in this use case.

2.3.7 Transaction Manager

Description

The Transaction Manager is intended to handle the transactions offered by the Triple Space API. Triple Space provides transactional support via the further extended API which allows for the creating and starting of transactions. These can be realized within the kernel by activating the Transaction Manager and use the further extended API to assert that all operations taking place within a given transaction display ACIDity properties.

The started transactions will be logged in the Transaction Manager. Subsequent operations are passed over the API within that transaction. Changes to the storage will have to be handled transactionally within the RDF storage layer, which is no

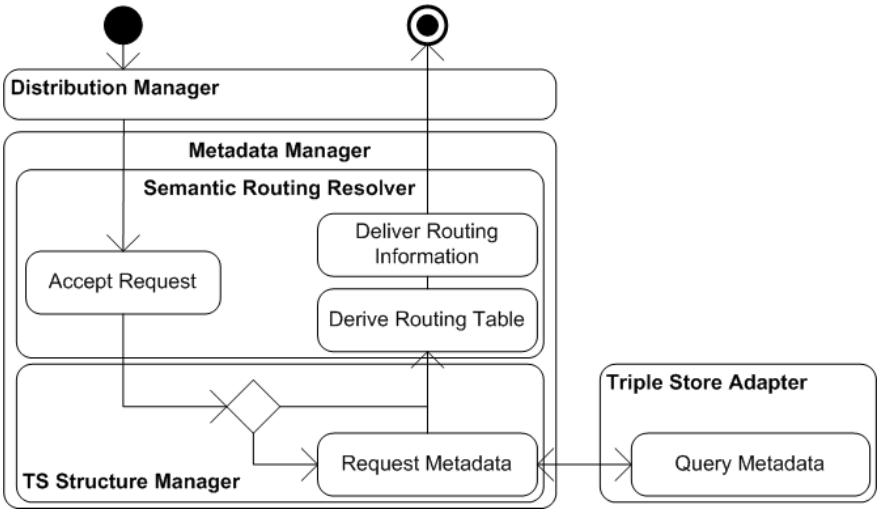

Figure 2.8: Use case 3: querying of metadata

problem as the ORDI framework supports transactions. Importantly, in the case of a rollback, all components will need to take care of possible compensation actions.

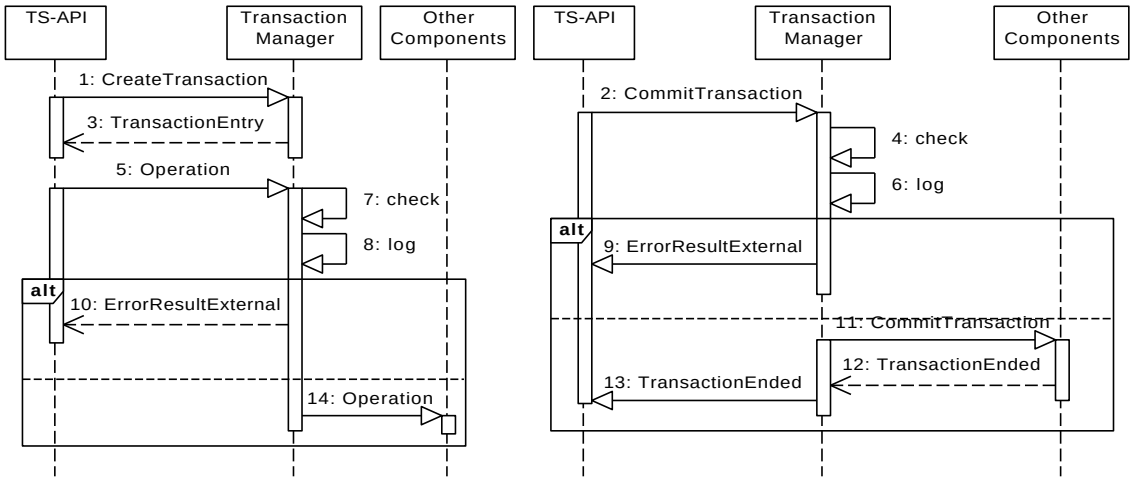

Figure 2.9: Transaction Manager interaction

Figure 2.9 illustrates the exchange of entries among the Transaction Manager and the rest of the components. The first operation to trigger the Transaction Manager is *createTransaction()*. A transaction is registered in the Transaction Manager with a (component internal) unique identifier.

Every operation, which is executed within a transaction, is logged by Transaction Manager in a "transaction log". To accomplish this the Transaction Manager monitors the operations on the integration space. When an operation is performed transactionally, the Transaction Manager checks whether the transaction id is valid[2] and adds the operation to the transaction log. If the client refers to an invalid transaction identifier an error is raised.

[2]A transaction is valid, if it has been created by the Transaction Manager but not yet commited or rolled back; otherwise it is invalid.

As the basic principle, all operations on the RDF storage which take place as an effect of the (kernel internal) processing of one of the operations within the the TS-API transaction, have to be executed within a corresponding ORDI transaction.

Use Cases

In the following, the three relevant use cases are presented: create the transaction, transactionally perform an operation, and commit or rollback, respectively, the transaction (see Figure 2.10).

Use case: Create transaction	
Description	A client issues the request to create a new transaction.
Preconditions	The system bus and the Transaction Manager are running and the Transaction Manager has access to the system bus.
Normal flow	The Transaction Manager creates a unique identifier for the transaction and writes this identifier into the system bus for further use by the TS-API component.
Alternative flow	If the Transaction Manager looses connection to the system bus after getting the request, it tries to reestablish a connection until it is shut down or the connection is successfully established.

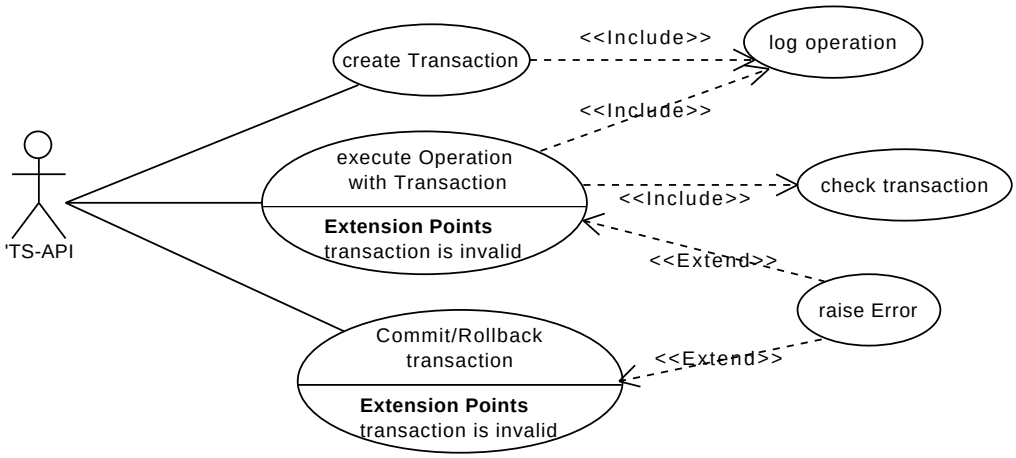

Figure 2.10: Transaction Manager use cases

Use case: Execute operation within a transaction	
Description	A client issues an operation within an active transaction.
Preconditions	The transaction has already been created and started.
Normal flow	The Transaction Manager consumes the request and checks whether the transaction is valid. If it is valid a log entry is added to the transaction log and the request is written back to the system bus.
Alternative flow	If the transaction is not valid an error is raised and the error information is written to the system bus.

Use case: Commit/rollback transaction	
Description	A client issues the request to commit or rollback a transaction.
Precoditions	The transaction has already been created and started.
Normal flow	The Transaction Manager consumes the request from the system bus and checks the transaction identifier. If the transaction is valid a log entry is added to the transaction log and the transaction entry informs all components which have to react on the commit or rollback of a transaction via the system bus.
Alternative flow	If the transaction is not valid an error is raised and the error information is written to the system bus.

2.3.8 Security Manager

The Security Manager (SM) is the kernel component in charge of checking the identity of clients and giving them appropriate permissions. Clients interact with a kernel by means of APIs, which are implemented both at the client and kernel side. Through the APIs the client sends requests to the kernel, such as `out` and `rd` requests, targeted at certain triplespaces.

Client requests need to be authorized by the Security Manager in order to be admitted in the internal, protected core of the kernel, where internal (trusted) components handle the requests and produce the corresponding responses. To ensure this separation, we will use separate spaces in the system bus for incoming unproofed operations, proofed operations (by the Security Manager) and the outgoing responses to those operations. These separate spaces are referred to as "areas".

Hence, incoming client requests received through the APIs are published in an *inbound area* of the system bus, where they are picked up by the Security Manager. Authorized requests are then propagated to an *internal area* of the integration bus. Authorization decisions are driven by the security policies configured by the owners of the triplespaces managed by the kernel. The approach is role-based, i.e. policy rules assign or deny certain permissions on the basis of the *security roles* assigned to clients. On the other hand, these roles are assigned on the basis of *attribute assertions* issued by trusted third parties and provided by the client. The client can retrieve these assertions by contacting one or more remote attribute providers using the SAML protocols and data format; the API component then propagates these assertions to the Security Manager inside the kernel. Attribute providers are external entities, and not part of TripCom infrastructure: how the client contacts an attribute provider and retrieves his attribute assertions is out of scope. Attribute providers play a role similar to traditional certification authorities.

Use cases

We can identify three different use cases for the Security Manager:

- *authentication*, i.e. performing authenticity checks on clients' credentials and other supplied information;

- *trust and attribute mapping*, i.e. filtering clients' supplied information (identity attributes) based on trust relationships with asserting parties, and mapping these attributes to roles for access control;
- *access control*, i.e. taking access control decisions based on roles.

The separate functionalities would be performed thus:

1. Authentication: the Security Manager receives the client certificate and assertion(s), and checks if they are valid and if the client can claim their possession. This involves checking the Attribute Provider signature on the assertion and verifying if the client is the legitimate holder of that assertion (e.g. by checking a proof of possession of a private key). The security context is initialized after this check.
2. Attribute trust: the Security Manager takes the client identity assertions and, basing on the Attribute Provider from which they come, decides whether to trust them or not, or to trust them only partially (i.e. only some attributes), and updates the security context.
3. Attribute mapping: the Security Manager takes the trusted attributes that result from the previous phase and maps them into roles that are relevant for the access control policy, updating the security context.
4. Access control decision: the Security Manager decides whether to authorize or not the request, basing on the client's roles and the access control policy of target space(s). If the operation is a "rd", the Security Manager also includes the list of subspaces of the target space in which the operation is authorized[3]. If the request is authorized, the operation is moved to the internal area of the kernel.

2.3.9 Web Service Registry

The Web Service Registry component is responsible for the storage of Web service descriptions (WSDL) and Semantic Web service descriptions (WSML) typically for its later discovery and retrieval making use of the Web Service Discovery component (see section 2.3.10).

The Web Service Registry component exposes a Web service API for its proper invocation by external clients to provide access to the registry, to look up Web Services described therein.

More in depth information about the design and implementation of the Web Service component can be found in D4.3 - "Methodology for adopting Semantic Web Services in a Triple Space environment" ([1]).

Use Cases

In this section we envision typical cases and scenarios where the Web Service Registry component will be used. The identification of these scenarios will allow us to identify

[3]This is not needed for "out" operations, since they cannot span on subspaces. See deliverable D5.2 for details.

the actors and roles involved and even the relations amongst the distinct use cases identified.

The use cases identified for the Web Service Registry component are depicted in Figure 2.11.

In these use cases we identify 3 roles:

1. **Submitting Organization:** An organization or part of it that wants to store information in the registry (typically about the services it offers).
2. **Content Submitter:** Person in charge of registering information in the registry in the name of a Submitting Organization.
3. **Registry Operator:** The registry administrator.

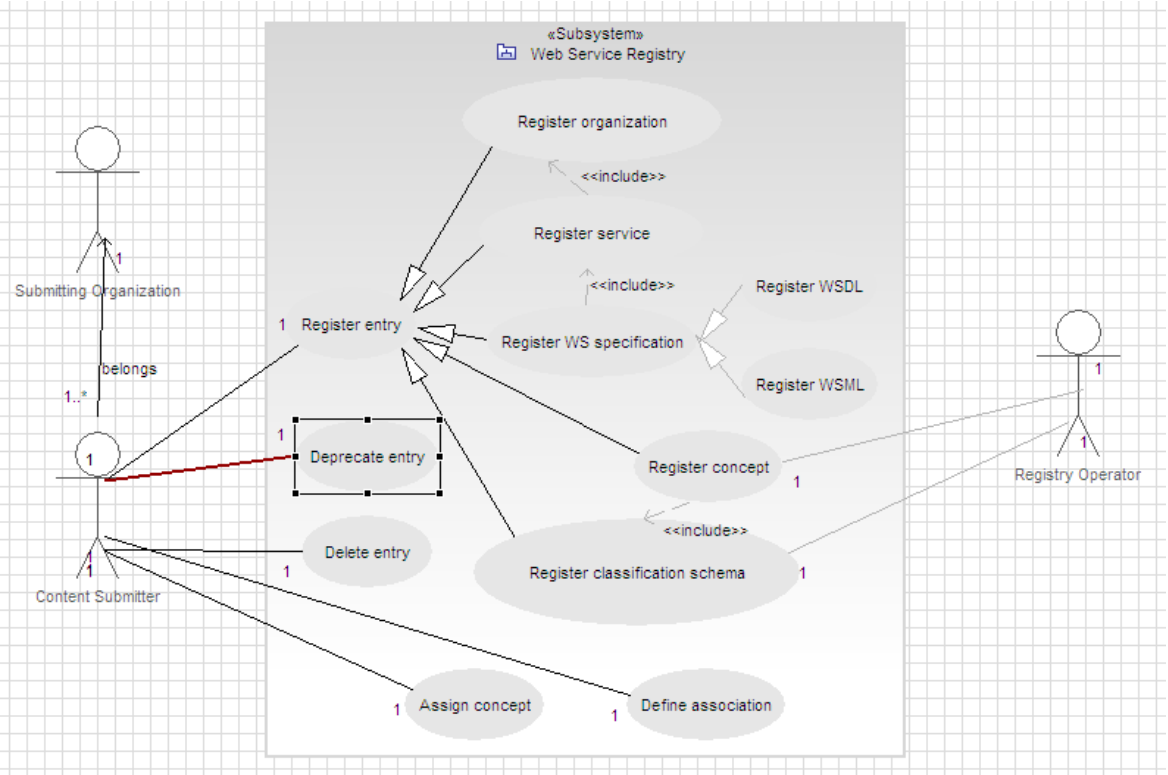

Figure 2.11: Use cases for the Web Service Registry

We detail two of the most significative use cases in Tables from 2.23 to 2.25.

Use case: Register entry	
Description	Registers a new entry in the Registry
Pre-conditions	Information about registry entry available
Post-conditions	Entry registered or error
Normal flow	1. Validate the information susceptible to be registered 2. Translate the information to RDF 3. Use the TS API to store the information as RDF triples in a space 4. Inform the client about the success of the registration operation
Error situations	- The data passed is insufficient or erroneous - The submitter is not authorized to register the information in the registry
Error flow	Send error to the client refusing the registration of the information and informing of the errouneous situation

Table 2.23: Use case: Register entry

Use case: Assign concept	
Description	Assigns a concept to a registry entry
Pre-conditions	Registry entry and concept available
Post-conditions	Concept assigned to registry entry or error
Normal flow	1. Validate the information susceptible to be registered 2. Translate the information to RDF 3. Use the TS API to store the information as RDF triples 4. Inform the client about the success of the registration operation
Error situations	- The data passed is insufficient or erroneous - The submitter is not authorized to register the information in the registry
Error flow	Send error to the client refusing the registration of the information and informing of the erroneous situation

Table 2.24: Use case: Assign concept

Use case: Delete entry	
Description	Deletes an registry entry from TripCom
Pre-conditions	Entry previously registered
Post-conditions	Entry deleted or error
Normal flow	1. Translate the information to RDF 2. Use the TS API to remove the RDF triples 3. Inform the client about the success of the deletion operation
Error situations	- The data passed is insufficient or erroneous - The submitter is not authorized to delete the information in the registry
Error flow	Send error to the client refusing the deletion of the information and informing of the errouneous situation

Table 2.25: Use case: Delete entry

2.3.10 Web Service Discovery

When a Web service client wants to invoke a Web service for achieving a given goal, the client may know the desired Web services, or may have no *priori* knowledge about the potential Web services which can meet this requirement. In the latter case, the Web service discovery component needs to provide support for locating potential and desired Web services. There are mainly two kinds of approaches for Web service discovery: WSMX and UDDI. The request may be in WSML format or in UDDI methods. They need to be translated into RDF. The translation shall be supported by *Adapters*.

Use Cases

In this section, we present typical use cases for Web service discovery. Before presenting use cases, we identify the actors (or roles) involved as below:

1. ***Discoverer***: The client (or a Web service) triggeres service discovery with a request message.
2. ***Goal Administrator***: The goal administrator for goal operations.

A client passes a Web service discovery request to this component. The client may block until a response is returned. A response is either a set of Web services matching this request, or an error message. The use cases for Web service discovery are presented in Tables from 2.26 to 2.28.

Use case: Discovery Web Services in WSDL	
Description	Discovery of Web services for a given goal where the goal is described in WSDL format
Pre-conditions	Goal message in WSDL format
Post-conditions	1. Result is returned 2. The goal and the result are stored in the Triple Space
Normal flow	1. Validate the goal 2. Translate the goal format from WSDL to RDF 3. Send query via TS API to query available Web services registered in Triple Space for this goal. 4. Call ***Store discovered result with a goal*** to store the goal and the result 5. Inform the goal administrator about storing (or updating) the discovered result for this goal 6. Translate the result format from RDF to WSDL 7. Inform the discoverer about the result of this goal
Error situations	1. The goal or the result is insufficient or erroneous 2. The goal administrator is not authorized to perform this operation 3. The discoverer is not authorized to perform this operation
Error flow	Send error to the discoverer refusing this operation with error message for specifying the cause of failure

Table 2.26: Use case: Discovery Web Services in WSDL

2.3.11 Web Service Invocation

Web service invocation aims at specifying a process for achieving a functionality provided by a Web service over a Web service transport. The Web service invocation component is responsible for the execution of a service.

Use case: Discovery Web Services in WSML	
Description	Discovery Web services for a given goal where the goal is described in WSML format
Pre-conditions	Goal message in WSML format
Post-conditions	1. Result is returned 2. The goal and the result are stored in Triple Space
Normal flow	1. Validate the goal 2. Translate the goal format from WSML to RDF 3. Send query via TS API to query available Web services registered in Triple Space for this goal. 4. Call ***Store discovered result with a goal*** to store the goal and the result 5. Inform the goal administrator about storing (or updating) the discovered result for this goal 6. Translate the result format from RDF to WSML 7. Inform the discoverer about the result of this goal
Error situations	1. The goal or the result is insufficient or erroneous 2. The goal administrator is not authorized to perform this operation 3. The discoverer is not authorized to perform this operation
Error flow	Send error to the discoverer refusing this operation with error message for specifying the cause of failure

Table 2.27: Use case: Discovery Web Services in WSML

Use case: Store discovered result with a goal	
Description	Store the result of a goal discovery in Triple Space for later reuse
Pre-conditions	1. Goal message 2. The result of query
Post-conditions	The result of discovery for a goal
Normal flow	1. Validate the goal and the discovered result. Pay attention that, the goal and the discovered result are represented in RDF format at this point 2. Use the TS API to store the result section of this goal in Triple Space 3. Inform the goal administrator about the result of this goal discovery operation
Error situations	1. The goal or the result is insufficient or erroneous 2. The goal administrator is not authorized to perform this operation
Error flow	Send error to the goal administrator refusing this operation with error message for specifying the cause of failure

Table 2.28: Use case: Store Discovered Result with a Goal

TripCom supposes to support *Remote Procedure Calls* style Web service interactions only, in which there is a single request-response message exchange. The discovered Web services agree with a given goal on a common interface.

Web service invocation in this section is *explicit service selection* approach specified in *D4.1: Architectural Integration of Triple Spaces with Web Service Infrastructures*. Regarding *implicit service selection* via WSMX, TripCOM is used as a storage for WSMX, so this scenario is out of the scope of Web service invocation.

Use Cases

The use cases for Web service discovery are presented in Tables from 2.29 to 2.30.

Use case: Web Service Invocation in WSDL	
Description	Invocation of a Web service for a given goal in WSDL format
Pre-conditions	Web service URI, Goal message in WSDL format
Post-conditions	1. The result is returned 2. The goal template is stored or updated in Triple Space
Normal flow	1. Validate this goal 2. Invoke Web service and get the result. 3. Translate the format from WSDL to RDF for the goal and the result. 4. Save the goal and the result into Triple Space. 5. Inform the invocator about the result of this invocation
Error situations	1. The goal or the result is insufficient or erroneous 2. The invocator is not authorized to perform this operation
Error flow	Send error to the invocator refusing this operation with error message for specifying the cause of failure

Table 2.29: Use case: Web Service Invocation in WSDL

Use case: Web Service Invocation in WSML	
Description	Invocation a Web service for a given goal in WSML format
Pre-conditions	Goal message in WSML format
Post-conditions	1. The result is returned 2. The goal template is stored or updated in Triple Space
Normal flow	1. Validate this goal 2. Invoke Web service and get the result. 3. Translate the format from WSDL to RDF for the goal and the result. 4. Save the goal and the result into TS space. 5. Inform the invocator about the result of this invocation
Error situations	1. The goal or the result is insufficient or erroneous 2. The invocator is not authorized to perform this operation
Error flow	Send error to the invocator refusing this operation with error message for specifying the cause of failure

Table 2.30: Use case: Web Service Invocation in WSML

2.3.12 Mediation Manager

The Mediator Manager[4] component is responsible for resolving the heterogeneities between different ontologies that describe similar problem domains for the proper understanding of the communicating partners.

More in depth information about the design and implementation of the Mediation Manager component can be found in D4.3 - "Methodology for adopting Semantic Web Services in a Triple Space environment" ([1]).

Use Cases

In this section we envision typical cases and scenarios where the Mediator Manager component will be used. The identification of these scenarios will allow us to identify the actors and roles involed and even the relations amongst the distinct use cases identified.

The use cases identified for the Mediator Manager component are depicted in Figure 2.12.

In these use cases we identy 2 roles:

1. **Mediation rules administrator:** Entity in charge of administering (registering, modifying, deleting) the mediation rules to be applied in the mediation process.

2. **Integrating middleware (Bus):** The integrating middleware or bus is the communication middleware used by Triple Space internal components to communicate. The Mediation Manager is called upon interception either of graph insertion or query evaluation.

These use cases are presented in depth in Tables from 2.31 to 2.33.

Use case: Manage mediation rules	
Description	Manages mediation rules in TripCom. These mediation rules will be used later for the proper mediation or translation of data that make use of distinct ontologies for similar problem domains
Pre-conditions	Mediation rules available
Post-conditions	Mediation rules managed or error
Normal flow	1. Check that the mediation rules that want to be managed are valid 2. Manage the mediation rules 3. Inform the client about the success of the operation
Error situations	- The mediation rules are not valid - The client is not authorized to manage mediation rules
Error flow	1. Send error to the client refusing the management of the mediation rules and informing of the errouneous situation

Table 2.31: Use case: Manage mediation rules

[4]Although the Mediation Manager is not part of the prototype to be implemented in the context of the TripCom project to validate the suitability of the Triple Space platform, it has been treated as the rest of the components that compose the Triple Space architecture.

Use case: Manage mediation WS	
Description	Manages mediation Web services so that they can be properly used by the mediation process
Pre-conditions	Information about mediation WS available
Post-conditions	Mediation rules deleted or error
Normal flow	1. Check that the mediation WS that want to be managed are valid 2. Manage the mediation WS 3. Inform the client about the success of the operation
Error situations	- The mediation WS are not valid - The client is not authorized to manage the mediation WS
Error flow	1. Send error to the client refusing the management of the mediation WS and informing of the errouneous situation

Table 2.32: Use case: Manage mediation WS

Use case: Mediate	
Description	Mediates information between distinct ontologies from a similar problem domain
Pre-conditions	Input data for the mediation available
Post-conditions	Data mediated or error
Normal flow	1. Check that the input data (data to be mediated, input ontology, output ontology) is valid 2. Mediate the data 3. Inform the client about the success of the operation returning the mediated data
Error situations	- The input data is not valid (there exists no mediation rules for the desired ontologies, etc.)
Error flow	1. Send error to the client refusing the mediation of the data and informing of the errouneous situation

Table 2.33: Use case: Mediate

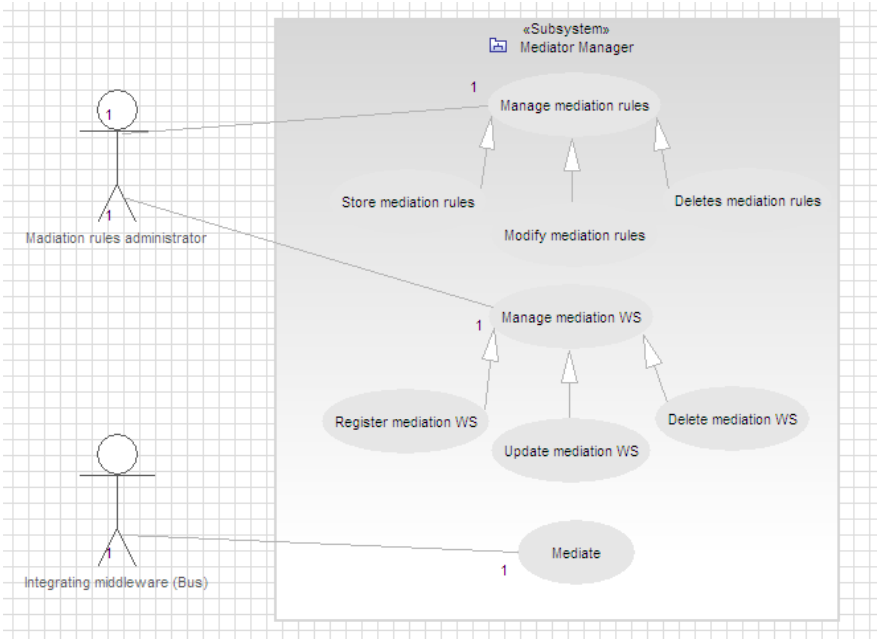

Figure 2.12: Use cases for the Mediation Manager

3 Component Integration

This chapter describes how the task of realizing the functionality of the TripCom kernel as a set of components has been approached, starting from a functional decomposition of the kernel into components[1] and a course functional description of the latter as described in [6]. The integration task includes (1) the ultimate fine grain functional specification of the individual components in interaction with (2) a detailed specification of the component interfaces, with respect to both the data transfered (data flow) and the sequence and conditions controlling these transfers (control flow).

3.1 Integration Middleware

The communication among the kernel components is realized on top of the integration middleware. As described in [6], the component integration in TripCom follows the space based paradigm. Instead of explicitly exchanging messages between the components or performing remote procedure calls, they communicate and coordinate themselves by simply writing and reading entries from/to a space based middleware. Synchronization is realized via accessing the data in the space via read operations which might block if the referred data does not yet exist. This leads to a decoupling of the components concerning time, space and reference and provides a fault tolerant, robust and load balanced way to integrate the components. In the following, the term *system bus* will refer to the kernel internal space (which is a tuple- rather triplespace) which connects the components with each other. An important feature of the space based integration approach is its support for increasing a kernel's ressources in a straightforward manner through running multiple component instances simultaneously; the individual instances do not influence each other, they do not even have to know about each other. If there is an increase of requests which can not sufficiently be handled by a single instance of a component, a second one can be created, running on a second machine, which directly supports the first one. The second instance may remotely access the system bus which remains hosted at the first machine.

As long as the component's interaction follows a request-answer pattern (i.e., a request—encoded in a data item—is taken from the system bus, then processed, and afterwards the answer is written back into the system bus), no further implementation effort is required, in particular, no code modifications are necessary. The capability of compensating a component's growing workload by incrementing its number of instances, i.e. by scaling "out" at the component level, is an important leverage of building a scalable (kernel) system.

The integration of the TripCom kernel components is also regarded as an excellent opportunity of performing a case study and proof of concept of space based integration of components of a distributed application. This is of particular interest in the context of a project which develops a (semantic) space technology.

Originally, as described in [6], the XVSM system has been chosen as the component integration platform for the TripCom kernel. This decision has been reconsidered in project month M20, when the JavaSpaces standard was selected instead. In a next

[1] Notice that that we do not a priori refer to "components" in the narrow sense of Component-Based Development (CBD); there are indications to view a kernel component rather as a "module", and a proper characterization of a kernel component as a piece of software remains for further examination.

step, based on a JavaSpaces subset which has been identified to be relevant for the development of the TripCom prototype, a lean "Integration API" has been specified (cf. `http://tripcom.sourceforge.net/integration_api`) as the API of the kernel components to the integration space, which in particular eliminates the bulk of JINI dependences existing in JavaSpaces. Indeed, following the requirements by partners of a stand-alone use of single components, i.e., their deployment in an environment different from TripCom, and consequently of relying on industrial standards only, this approach was abandoned in favour of pure JavaSpaces.

3.1.1 JavaSpaces

JavaSpaces[2] is an interface specification by Sun Microsystems which realizes the space based computing paradigm centered around the concept of distributed object spaces. The space represents an associative memory which can be accessed over a network. Processes which communicate via JavaSpaces, do not communicate directly with each other, instead communication takes place by exchanging objects via the space.

Objects in the space are represented as *tuples*, based on principles of the Linda programming language. In JavaSpaces, a process can *write*, *read* or *take* objects. When reading or taking an object, a *template* can be supplied which has to be matched. If there is no object existent which matches the template, the process waits until a matching object has been written. JavaSpaces also supports *notifications*. After a process has registered a notification, the space notifies the process whenever a new object has been written. The JavaSpaces specification defines the following methods.

- `Lease write(Entry entry, Transaction txn, long lease);`

  Writes an entry into the space. The entry can be written under an optional transaction. The lease time specifies the minimum time the entry shall be available in the space.

- `Entry read(Entry tmpl, Transaction txn, long timeout);`

  Reads one entry with an optional transaction from the space which matches the given template. If no entry is available the method blocks until an entry is available or the timeout expired.

- `Entry take(Entry tmpl, Transaction txn, long timeout);`

  Has the same semantic as the read method but additionally removes the read entry from the space.

- ```
  EventRegistration notify(Entry tmpl, Transaction txn,
                           RemoteEventListener listener,
                           long lease, MarshalledObject handback);
  ```

  Registers a notification which notifies the process whenever an entry is added which matches the template tmpl. The notification is performed by calling the notify() method of the RemoteEventListener class. The lease specifies the time the notification is valid and a handback object will be sent back to listener as part of the event notification.

[2] `http://www.sun.com/software/jini/specs/js2_0.pdf`
  ```

3.1.2 Blitz Implementation

In TripCom, the JavaSpaces implementation *Blitz*[3] is employed. This open source implementation has been developed by Dan Creswell. It has been choosen mainly because it offers the following features[4].

Configurable persistency mechanism Blitz offers a range of *storage models* which allow for configuring the persistency behaviour of the space. Thus, the best compromise between data-integrity and speed can be choosen.

Tools for browsing the content The *dashboard* which is distributed along with Blitz visualizes the space (operations, entries, ...) and its memory usage. This information supports the development, in particular the debugging process, and offers a way to monitor the interaction of the components.

Embeddable space This feature allows for the use of Blitz within the same Java Virtual Machine in which the application is running. If no distribution of the components is required, the embedded space results in a performance boost because no remote communication is necessary. Running the space within the same JVM offers a lot of possibilities of deploying the kernel without the need of changing the implementation of the components.

Despite these advantages, the TripCom implementation does not depend on Blitz. Since Blitz adheres to the standardized JavaSpaces and Jini[5] interfaces and services, the JavaSpaces implementation can be replaced without changes to the kernel components.

3.2 Implementation Plan

As described in chapter 2.1 the approach used for the first TripCom implementation is based on *Scrum*. Because of the geographical distribution of the project members the approach has been slightly modified. In the following, on overview of the concepts of Scrum wich have been employed in TripCom is given.

- A *backlog* is a list of features which will be implemented in the prototype. The representatives of the *supervising partners* (cf. Table A.1 in [6]) are responsible for preparing this list for their component.
- A *sprint* is a central concept of Scrum. It is a time period within which development takes place. During a sprint the component teams independently implement backlog items. A period of one month is regarded appropriate for the TripCom prototype implementation. For TripCom, a sprint lasts 2 weeks.
- A *sprint meeting* is a telephone (eventually skype) conference which takes place at the beginning of each sprint. All component representatives participate in this conference. They are responsible for defining a set of backlog items for their component prior to the telephone conference, jointly with their development

[3] `http://www.dancres.org/blitz/index.html`
[4] `http://www.dancres.org/bjspj/docs/docs/blitz.html`
[5] `http://www.jini.org`

team. This set of item represents the implementation plan for the component for the next sprint.

Dependencies between backlog items of different components have to be discussed and an agreement of the *priority* of backlog items (and hence the decision, which one will be implemented first) has to be obtained. The outcome of the sprint meeting is an ordered list of backlog items to be implemented during the upcoming sprint.

In addition, each participant gives a short summary of the last sprint, which in particular includes both the backlog items which have been implemented and the ones which have not. Other issues to be discussed during the sprint meeting are the reasons for not implementing a backlog item, the implications on the further implementation task, problems, and suggestions.

- *Weekly reports* are used to track the progress of the implementation and to check if the implementation aligns with the backlogs. These reports contain a brief description of the progess of the last week. Especially completed backlog items and problems which occured should be mentioned.

The organization of team internal development is not specified within Scrum.

Component development for the first TripCom implementation has been subdivided into three iterations of about one month and a subsequent integration phase. Prior to the first iteration and overlapping with it, the work on the detailed component interfaces took place. The latter, highly collaborative, task has been centered around a number of documents managed via the project's shared repositories: the definition of JavaSpace Entry classes, the dependencies among the components as expressed in the component-component matrix and sequence diagrams of typical component interactions (cf. Section 3.4). Besides the collaborative editing of specification documents and intensive E-mail communication, the distributed software development process has been supported by regular skype conferences of involved work package leaders, the weekly reporting scheme. The integration task in particular benefits from the availability of the actual code of all components at SourceForge (see below).

3.3 Development Tools and Platform

The first TripCom implementation is supported by well established software development tools and is carried out over a publicly accessable software development platform.

Maven is a build manager for Java applications. It can manage the build process of the source code, download and install third party dependencies and provides a mechanism which generates project reports. An advantage of maven is that it defines a clear directory structure for sources, test sources and resource (configuration) files.

Each Triple Space component is a separate Maven project which allows the component developer for implementing and testing the component independently of others. Maven supports the creation of a project which has sub-projects, hence it is possible to compile and test components all at once (see `http://www.maven.org`).

JUnit is a framework for testing java application with smooth integration in the Maven build process (see `http://www.junit.org/`).

JCoverage can be used for coverage analysis of tests. The analysis yields the degree up to which the source code has been tested. As JUnit, JCoverage is well integrated in the Maven build process (see `http://www.jcoverage.com/`).

For coding, managing and publishing the TripCom implementation, the software development management system **SourceForge** (cf. `http://sourceforge.net/index.php`) is employed.

3.4 Component Interfaces

As described in Section 3.1, the components are integrated using the JavaSpaces technology. They communicate with each other by writing objects into and reading objects from the space. In the JavaSpaces terminology, these objects are called "entries". This section presents all the entries exchanged by the kernel components.

The data flow among the components is specified by means of a matrix (Table 3.1). Each line as well as each column of this "component-component" matrix represents a kernel component, i.e. each entry in this matrix is identified by an on ordered pair (c_A, c_B) of components. A matrix entry (c_A, c_B) contains exactly the JavaSpace entry classes[6], instances of which are written into the system bus by component c_A *and* which are read, or taken, respectively, from the system bus by component c_B. In addition, a detailed description of each entry is given (Section 3.4.1).

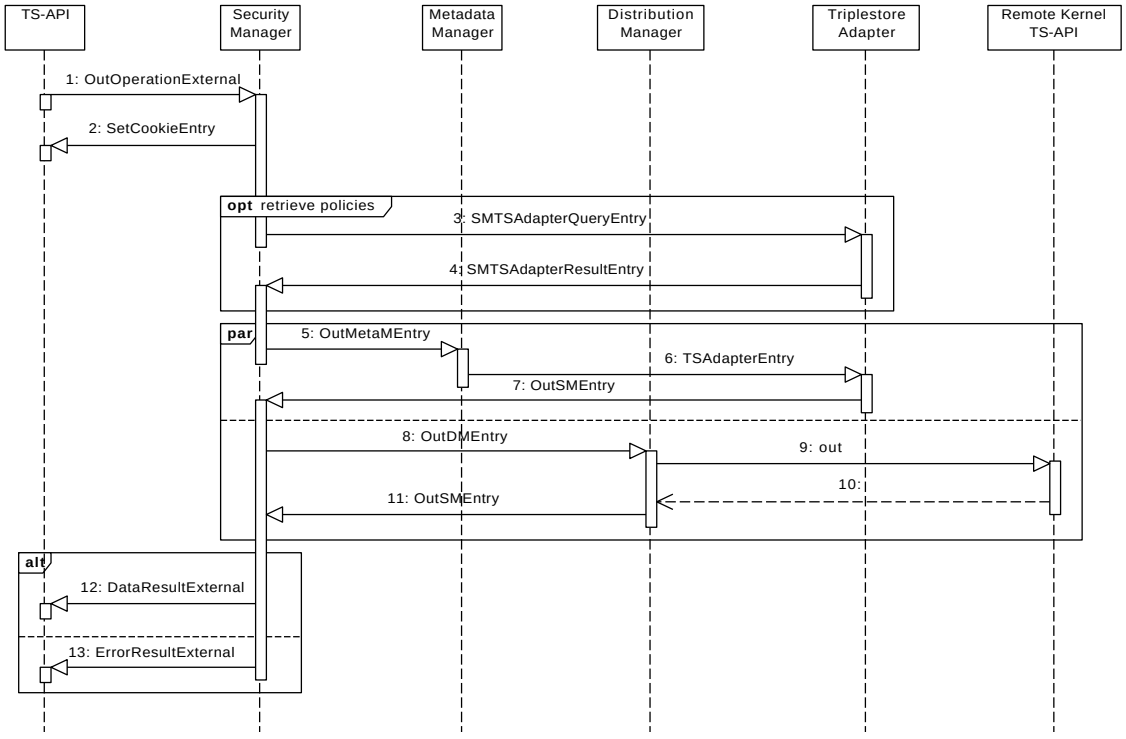

Figure 3.1: Processing of a TS API out() operation

As the component-component matrix does not specify also the prescribed patterns of interaction of components, these are depicted in additional sequence diagrams, namely for the execution of a TS-API out() operation (Figure 3.1) and rd() operation

[6]classes implementing the (marker) interface `et.jini.core.entry.Entry`

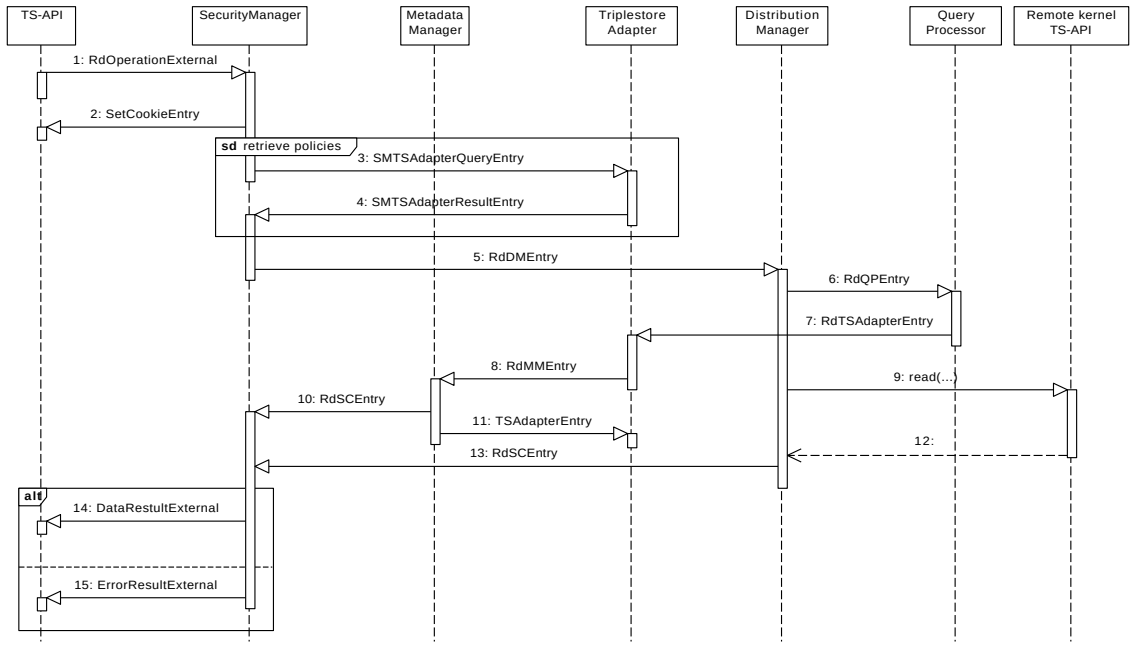

Figure 3.2: Processing of a TS API rd() operation with space URL

(Figure 3.2) operation, respectively. These sequence diagrams illustrate the data flow in the form of messages between the components.[7] Arrows are marked with the entries exchanged. Notice that data is not sent directly from one component to the other, but via the integration space (usually, by executing a combination of a write() operation and its corresponding take() operation). In order to avoid the otherwise overburdening of the diagrams, the integration space itself is not shown as a separate instance. Also the *dual* nature of the component interfaces as (1) interfaces to the integration space, and (2) interfaces to other components, respectively, may justify this abbreviation.

Messages 1 to 4 are very similar in both diagrams. A client issues an operation and the TS API writes an entry into the integration space which represents this operation. We call this entry the internal representation of the TS API operation. The execution of every TS API operation is checked by the Security Manager. It sends a cookie, which is used in subsequent kernel-client communication, to the TS API component, and (optionally) queries security policies from the Triplestore Adapter.

The following messages differ in the two diagrams. In case of a TS API out() operation there are two possibilities. If the target space of the request is managed by the local kernel, the Security Manager sends the request to the Metadata Manager in order to update the meta data. Afterwards the request is forwarded to the Triple Store Adapter, and the triples are stored in the RDF store. As opposed, if the target space is not managed by the local kernel, the Security Manager sends the request to the Distribution Manager which then forwards the request to the kernel which manages the space. In both cases the Security Manager waits for an acknowledgement. When

[7]Obviously the question whether we thereby leave the space based paradigm, deserves further discussion. As an elementary observation, the TripCom kernel components are designed in such a way that they jointly ensure the overall kernel functionality. This requirement induces an inevitable minimal coupling, which strictly speaking contradicts the strong decoupling principle of space based computing. This question also relates to the question on the nature of the TripCom kernel components (see the footnote in Section 3.1).

this acknowledgement has been received the Security Manager passes this information to the API. In the case of an error the according error information is returned.

In the case of a TS API rd() operation the request is forwarded to the Distribution Manager. The latter issues requests on remote kernels if necessary and passes the request to the Query Processor which processes the query locally. Afterwards the Query processor sends the rd() request to the Triple Store adapter, where the actual query for the data takes place. The result of the query is forwarded to the Metadata Manager for updating the meta data. The Security Manager has to wait for two entries, the first being the result of the local query and the second being the result of the remote query issued by the Distribution Manager. The Security manager combines these two result sets and passes the read triples back to the TS API.

3.4.1 Entry descriptions

This section gives a detailed account of the entries conjoining the kernel components. Each entry class is described, together with the names and types of all of its public fields and their descriptions.

Entry and fields	Field type	Description
ClientInfo		Information about a client, including its security role securityData field (may carry additional security information associated to the request, to be finally intercepted by the Security Manager together with the response).
roles	Set<String>	The roles associated to the client
clientID	String	The id of the client, if authenticated
securityData	String	An opaque field carrying security information
DataResultExternal		Data result of a rd operation to be returned to the TS API.
result	Set<Set<Statement>>	The result of the operation
operationID	long	A unique identifier of the operation
ErrorResultExternal		Error result of a TS API or Management API operation.
errorNumber	int	The numeric representation of the error
errorDescription	String	A textual description of the error
operationID	long	A unique identifier of the operation
ManagementDataResultExternal		Data result of a management operation to be returned to the Management API. The object depends on the operation, e.g., a RDF graph is represented as a set of triples.
result	Set<Object>	The result of the management operation
operationID	long	A unique identifier of the operation
ManagementOpExternal		Representation of a Management API operation.
operation	ManagementOperation	The type of operation requested
opParameters	List<Object>	The parameters of the operation
SecurityInfo	SecurityInfo	Security related data
operationID	long	A unique identifier of the operation
MetaMQueryEntry		A query for the metadata associated to a given space, sent to the Metadata Manager. The actual query type depends on the kind enumeration parameter: "Subspaces" queries request for the URLs of direct children of a target space.
operationID	long	A unique identifier of the operation

timeout	Timeout	The timeout information of the operation
space	URI	The space whose metadata should be retrieved
MetaDataKind	kind	The sort of metadata requested
MetaMResultEntry		The response to MetaMQueryEntry, returned by the Metadata Manager. The actual type of result depends on the query type as specified by the kind field: "Subspaces" queries return Set<URL>.
operationID	long	A unique identifier of the operation
timeout	Timeout	The timeout information of the operation
space	URI	The space whose metadata should be retrieved
result	Object	The result of the query
MgmtMetaMEntry		An authorized TS management operation, sent by the Security Manager to the Metadata Manager.
operation	ManagmentOperation	The type of operation requested
opParameters	List<Object>	The parameters of the operation
clientInfo	ClientInfo	Information about the client
operationID	int	A unique identifier of the operation
OutDMEntry		Internal representation of an out operation.
data	Set<TripleEntry>	The triples written by the client.
space	URI	The space the triples are written too, as indicated by the client
timestamp	Long	The time when the operation was issued
clientInfo	ClientInfo	Information about the client
operationID	long	A unique identifier of the operation
transactionID		The identifier of the transaction if one is used
OutMetaMEntry		This entry is equals to OutDMEntry.
OutOperationExternal		Representation of a TS write operation. Includes security-info that carries information required by the security manager to perform authentication and access control. SecurityInfo would be a superclass for either identity assertions ; in both cases, it may contains an X509v3 client certificate, used for authentication purposes.
data	Set<TripleEntry>	The triples which shall be written
space	URI	The URI of the target space
security-info	SecurityInfo	Security related data
operationID	long	A unique identifier of the operation
transactionID	URI	The identifier of the transaction if one is used
timestamp	Long	The time when the operation was issued
OutSMEntry		Confirmation entry for a completed write operation, sent by the TS Adapter to the Security Manager.
operationID	Long	A unique identifier of the operation
clientInfo	ClientInfo	Information about the client
RdDMEntry		Internal representation of a read request.
template	Template	The query of this read operation
space	URI	The space from which the entry shall be read
timestamp	long	The time when the operation was issued
type	ReadType	Indicates which type of read operation this is
clientInfo	ClientInfo	Information about the client
operationID	long	A unique identifier of the operation
transactionID	URI	The identifier of the transaction if one is used
subspaces	Set<URL>	Additionally allowed identifiers for subspace to be queried
qpData	QueryProcessingData	Contains control data for the Query Processor
toFromReasoner	Boolean	Indicates if the entry has to be processed by the Query Processor

	ReadType	Enumeration of all possible read operations.
	IN	Read triples and remove them from the space
	READ	Read triples. The triples stay unchanged
	READMULTIPLE	As READ but multiple matching triples are returned
	INMPULTIPLE	As IN but multiple matching triples are returned
	NOTIFY	Used for the creation of a notification
RdMetaMEntry		Response from TSAdatper to SPARQL query request.
data	Set<TripleEntry>	The returned triples matching the user template
space	URI	The URI of the target space
tripleset	Set<URI>	The triplesets associated with the data
timestamp	long	The time when the operation was issued
type	ReadType	Indicates which type of read operation this is
clientInfo	ClientInfo	Information about the client
operationID	long	A unique identifier of the operation
transactionID	URI	The identifier of the transaction if one is used
qpData	QueryProcessingData	Contains control data for the Query Processor
toFromReasoner	Boolean	Indicates if the entry has to be processed by the Query Processor
RdOperationExternal		Representation of a TS read operation. Template abstracts from any query type and can be subclassed to indicate query language. Timeout indicates the timeout given on the read operation and should be summed with the timestamp to determine a system time from which the operation should no longer be processed.
query	Template	The query of this read operation
space	URI	The URI of the target space
security-info	SecurityInfo	Security related data
operationID	long	A unique identifier of the operation
transactionID	URI	The identifier of the transaction if one is used
timestamp	Long	The time when the operation was issued
timeout	Timeout	The timeout information of the operation
type	ReadType	Indicates which type of read operation this is
qpData	QueryProcessingData	Contains control data for the Query Processor
toFromReasoner	Boolean	Indicates if the entry has to be processed by the Query Processor
RdQPEntry		Internal representation of a read operation.
template	Template	The query of this read operation
space	URI	The URI of the target space
timestamp	long	The time when the operation was issued
type	ReadType	Indicates which type of read operation this is
clientInfo	ClientInfo	Information about the client
operationID	long	A unique identifier of the operation
transactionID	URI	The identifier of the transaction if one is used
qpData	QueryProcessingData	Contains control data for the Query Processor
toFromReasoner	Boolean	Indicates if the entry has to be processed by the Query Processor
RdSCEntry		The distribution manager writes this entry for the answers that are not deliverd by the local kernel .
data	Set<TripleEntry>	The returned triples matching the user template
space	URI	The URI of the target space
tripleset	Set<URI>	The triplesets associated with the data
clientInfo	ClientInfo	Information about the client
operationID	long	A unique identifier of the operation
transactionID	URI	The identifier of the transaction if one is used

qpData	QueryProcessingData	Contains control data for the Query Processor
toFromReasoner	Boolean	Indicates if the entry has to be processed by the Query Processor
RdTSAdapterEntry		Request to read data from TSAdapter with SPARQL query.
query	String	A SPARQL query derived from the client's template
space	URI	The URI of the target space
timestamp	long	The time when the operation was issued
type	ReadType	Indicates which type of read operation this is
clientInfo	ClientInfo	Information about the client
operationID	long	A unique identifier of the operation
transactionID	URI	The identifier of the transaction if one is used
qpData	QueryProcessingData	Contains control data for the Query Processor
toFromReasoner	Boolean	Indicates if the entry has to be processed by the Query Processor
RdWithoutSpaceDMEntry		Linda rd Operation without Space URL.
template	Template	The query of this read operation
timestamp	long	The time when the operation was issued
type	ReadType	Indicates which type of read operation this is
clientInfo	ClientInfo	Information about the client
operationID	long	A unique identifier of the operation
transactionID	URI	The identifier of the transaction if one is used
subspaces	Set<URI>	Additionally allowed identifiers for subspace to be queried
qpData	QueryProcessingData	Contains control data for the Query Processor
toFromReasoner	Boolean	Indicates if the entry has to be processed by the Query Processor
SecurityAssertionsInfo		Security data structure which extends SecurityInfo with a set of SAML assertions, encoded as strings.
certificate	String	The optional client's certificate
assertions	Set<String>	The attribute assertions submitted by the client
SecurityCookieInfo		Security data structure which extends SecurityInfo with a security cookie, encoded as a string, which carries an opaque ID identifying the security context associated to the client.
certificate	String	The optional client's certificate
cookie	String	The security cookie submitted by the client
SecurityInfo		Superclass for security-related data transmitted in client requests. This base entry may contain an optional X509v3 certificate, which is a binary object encoded as a Base64 string. Subclasses are SecurityAssertionsInfo and SecurityCookieInfo.
certificate	String	the optional client's certificate
SetCookieEntry		An entry returned as soon as possible by the Security Manager to the API, in order to set the security cookie of the client.
cookie	String	The security cookie to associate to the client
operationID	long	A unique identifier of the operation
SMTSAdapterQueryEntry		A read request submitted by the Security Manager to the TS Adapter, in order to retrieve a security policy.
id	long	Identifier to be used to collect the query results
sparqlString	String	The query submitted to the TSAdapter
SMTSAdapterResultEntry		A result to a SMTSAdapterEntry request, returned by the TS Adapter to the Security Manager.
id	long	Identifier is used to link the query results to query

data	Set<TripleEntry>	The result to the query
space	URI	not used
tripleset	Set	not used
Template		Representation of a query. Currently only Templates containing SPARQL queries exist but there may be other queries in the future.
TripleEntry		Entry class to represent a triple used in the triplespace; If you want to exchange a triple with another component please use this class, because it contains the logic to validate its conformance with RDF specification. You can find the full definition of the used types on: `http://www.openrdf.org/doc/sesame2/api/`.
subject	Resource	The subject of the RDF statement
predicate	URI	The predicate of the RDF statement
object	Value	The object of the RDF statement
TSAdapterEntry		Representation of a request to the Triple Store Adapter.
operationID	long	A unique identifier of the operation
data	Set<TripleEntry>	The triples which shall be written
space	URI	The space the triples are written too
annotations	Set<TripleEntry>	Metadata about the user data to be stored in the metadata space
clientInfo	ClientInfo	Information about the client

from / to	TS-API	Management API	Triple Store Adapter	Security Manager	Metadata Manager	Transaction Manager	Distribution Manager
TS-API				OutOperation-External, RdOperation-External			
Management API				Management-Operation-External			
Triple Store Adapter				OutSMEntry, SMT-SAdapter-ResultEntry	RdMetaMEntry		
Security Manager	DataResult-External, ErrorResult-External, SetCookie-Entry	Management-DataResult-External, ErrorResult-External, SetCookie-Entry	TSAdapter-Entry, SMTS-Adapter-QueryEntry		OutMetaM-Entry, Mgmt-MetaMEntry, MetaMQuery-Entry		OutDMEntry, RdDMEntry
Metadata Manager			TSAdapter-Entry	RdSCEntry, MetaMResult-Entry			
Transaction Manager			TSAdapter-Entry	RdSCEntry, MetaMResult-Entry			
Distribution Manager				OutSMEntry, RdSCEntry, RdTSAdapter-Entry			

Table 3.1: Component component matrix

4 First TripCom Implementation

In this section a detailed description of the installation and the deployment process of the TripCom implementation is given. The usage of the current prototype is illustrated with an example. Finally the realized functionality of the prototype is summarized and some subversion statistics are presented.

4.1 Starting and deploying the prototype

This section describes the required steps to download, compile and use the Triple Space prototype.

4.1.1 Required Software

In order to download and compile the prototype the following software has to be installed.

- Java Development Kit: JDK 1.5[1]. Please note that java 5 has to be installed because Blitz does not work with java 6.
- Maven software development tool: maven 2.0[2]
- Subversion version control system: subversion 1.4[3]

4.1.2 Dependencies

All dependencies are available in maven repositories therefore maven handles the download and the installation of them. The TripCom Maven repository is password protected. To use it, the username and the password have to be set in the Maven configuration files. Below is the required content of `$HOME/.m2/settings.xml`[4]. The username is *tripcom* and the password *TP!56tsc*. The url of the repository is `http://tripcom.sourceforge.net/repository`. In order to use this repository it has to be added to the pom.xml[5] configuration file.

```
<settings>
  <servers>
    <server>
      <id>TripComRepository</id>
      <username>tripcom</username>
      <password>TP!56tsc</password>
    </server>
  </servers>
</settings>
```

Required changes to pom.xml for using the repository.

[1] `http://java.sun.com/javase/downloads/index_jdk5.jsp`
[2] `http://maven.apache.org/`
[3] `http://subversion.tigris.org/`
[4] settings.xml is the configuration file of Maven.
[5] In TripCom it has already been added to all components.

```
<project>
  ...
  <repositories>
    <repository>
    <id>TripComRepository</id>
    <url>http://tripcom.sourceforge.net/repository</url>
    </repository>
    ...
  </repositories>
  ...
</project>
```

4.2 Blitz

In this section it is described how to install and start the Blitz JavaSpace. Blitz JavaSpace is a jini service, therefore jini has to be installed in order to run Blitz.

1. Download jini: `http://www.jini.org/wiki/Category:Getting_Started`
2. Install jini: Execute the installer and follow the instructions.
3. Download blitz: from `http://sourceforge.net/project/showfiles.php?group_id=126322&package_id=138181`. File: installer_pj_2_0-rc4.jar
4. Install blitz: `java -jar installer_pj_2_0-rc4.jar` and follow the instructions.

4.2.1 Configuration

The default configuration of Blitz uses network multicasts to find the reggie registry. When another registry is found Blitz will automatically distribute its entries if there is another Blitz instance running. So, if multiple independent kernel instances shall be hosted within one network the multicast lookup has to be turned of and singlecast lookup has to be used.

Since Blitz uses multicast lookup ad default the following lines have to be added to `config/blitz.config` to tell Blitz where it can find the jini registry (reggie).

```
...
import net.jini.core.discovery.LookupLocator;
...
org.dancres.blitz {
        ...
        initialLocators =  new LookupLocator[]
                               {new LookupLocator("jini://localhost")};
        ...
}
```

The TransactionManager (mahalo) also needs to know where it has to register at startup. Therefore the following lines have to be added to `conf/mahalo.conf`.

```
....
import net.jini.core.discovery.LookupLocator;

com.sun.jini.mahalo{
        ...
        initialLookupLocators = new LookupLocator[]
                                    {new LookupLocator("jini://localhost")};
}
```

In order to completly disable multicasts the jini registry has to be configured to not use any network interface for broadcasting. The following lines have to be added to `conf/reggie.conf`.

```
...
import java.net.NetworkInterface;

com.sun.jini.reggie{
        ...
        multicastInterfaces = new NetworkInterface[]{};
}
net.jini.discovery.LookupDiscovery{
        multicastInterfaces = new NetworkInterface[]{};
}
```

Blitz comes with an monitoring application wich is called “dashboard”. The dashboard uses multicast lookup to find the JavaSpace instance. In order to use the dashboard without broadcast lookup the hostname and the port have to be passed as arguments in the start script:

```
...
$JAVA_HOME/bin/java $POLICY -cp $CP \
  org.dancres.blitz.tools.dash.StartDashBoard \
  localhost:4160 $SPACE_NAME
```

4.2.2 Starting Blitz

Blitz can be started using the provided shell script. Blitz does not work with java 1.6. In the Blitz start script the home directory of the java virtual machine can be configured (`JAVA_HOME`). This variable has to point to a java 1.5 installation.

```
sh start.sh
```

4.3 Compile

This section describes how the kernel sources can be downloaded and compiled.

4.3.1 Obtaining the source

First, the source code has to be checked out from the sourceforge subversion repository[6].

[6] see `http://sourceforge.net/svn/?group_id=169344`

```
mvn co https://tripcom.svn.sourceforge.net/svnroot/tripcom
```

This command copies all tripcom related source code to the local working directory. A directory with the name "tripcom" will be created were the files are stored.

4.3.2 Compiling the source

All kernel components can be build with the command shown below. The command has to be executed in the directory `tripcom/trunk/`.

```
mvn clean package assembly:assembly
```

By default, Maven executes the test cases of all components after compiling. If the execution of the test cases shall be skipped `-Dmaven.test.skip` has to be appended. After a successfull compilation Maven creates a zip (tripcom-${version}-bin.zip) archive in the directory "target". The zip archive contains the following files and directories:
/libs all third party libraries
/bin the kernel components
/conf configuration files
start.sh, start.bat start script for unix and Windows

4.4 Example

The example source code depicted in figure 4.1 illustrates the use of the Java API Implementation. The installation and startup process of the kernel is described in section 4.1 which has to be followed in order to try the example.

First an instance of the Core API is created. The constructor requires a String which contains the security information of the client. In this example the variable `CERTIFICATE` contains the clients certificate. Afterwards, on RDF triple is created. The following code snippet depicts the N3 notation of the triple.

```
@prefix dc: <http://purl.org/dc/elements/1.1/>.
<http://example.com/book/book1>
  dc:title "SPARQL Tutorial"
```

This triple is written into the space with the url `tsc://www.example.com:4321`. After the triple has been written successfully, a query is created which will be used to read the previously written triple. A very simple triple is used which matches every available triple in the space. Please note that with the Core API only one matching triple will be returned. In order to get all triples the `rdmultiple` method from the Extended API (see 2.2) has to be used.

The read operation is issued at the same space as the write operation and a timeout of 3 seconds is used. The timeout indicates that the space shall try to find matching entries for at least three seconds. Finnally, the triple which has been read is printed to the standard output.

```
import org.openrdf.model.*;
import org.tripcom.api.ts.*;

public class TSTest {

 public static void main(String args[]){
  // Create instance of the Triplespace API component
  CoreAPI tsapi = new CoreAPIImplementation(CERTIFICATE);

  // Create a the test triple
  ValueFactory factory = new ValueFactoryImpl();
  URI subject = factory.createURI("http://example.com/book/book1");
  URI predicate = factory
                    .createURI("http://purl.org/dc/elements/1.1/title");
  Literal object = factory.createLiteral("SPARQL Tutorial");
  Statement statement = new StatementImpl(subject,
                                          predicate,
                                          object);

  // Write the triple into the space
  tsapi.out(statement, new URIImpl("tsc://www.example.com:4321"));

  // Create the SPARQL query and read the triple from the space
  String query = "SELECT * WHERE { ?x ?y ?z . }";
  Set<Statement> result = tsapi.rd(query, new URIImpl(
                                    "tsc://www.example.com:4321"),
                                    30000);
  System.out.println("Result read with space url: " + result);
 }
}
```

Figure 4.1: Triple space example

4.5 Realized functionality

As depicted in the example in section 4.4 the current implementaion of the triple space prototype supports the writing and reading of triples. In order to realize this functionality, the Core API (section 2.2) and the extended API (section 2.2) are supported. The prototype can be used as stand alone service whereby each component is running in its own Java Virtual Machine. As described in section 3.1 integration platform a space based computing middleware has been selected. Therefore, the communication between the kernel components takes by means of exchanging information via a JavaSpace middleware. There are currently two ways to use the kernel. The nativ Java API and SOAP messages. For the communication between kernel instances SOAP messages are used and a client can choose freely between these two alternatives. The use of the Java API is depicted in figure 4.1. Alternatively the request can be sent to the kernel in form of SOAP message (for example by using AXIS[7]).

The tables 4.1, 4.2 and 4.3 summarize the supported functionality of the components in respect to the core, extended and further extended API. Fields marked with *Y* depict that the component supports the function, *N* means that the component does not support the function for this prototype and *X* means that the component does not have to participate in order to execute the function. From these tables, the currently supported triple space configurations (section 1.3.1) can be read off.

	out	rd	rd without Space
DistributionManager	Y	Y[8]	Y[8]
MetadataManager	Y	Y[10]	Y
SecurityManager	Y	Y	N[12]
TS Adapter	Y	Y	Y
TS API	Y	Y	Y
QueryProcessor	N	Y	N

Table 4.1: Supported functionality of the core API.

4.6 Statistics

As described in section 3.2 sourceforge is used as implementation platform for the prototype. In this section some statistics of implementation progress are summarized. Figure 4.2 depicts the usage of the subversion repository in the last 12 months and table 4.4 contains the absolute values since november. The x-axis indicates the time months. The left vertical axis depicts the transactions issued in this month and the right one indicates the files per month scaled in thousands.

[7] `http://ws.apache.org/axis/`
[8] only simple pattern query.
[9] will be implemented in the last iteration for the extended configuration of Triple Space
[10] no metadata necessary.
[11] supported, but only via reduction to the simple/single case via other componentes
[12] no security check for this prototype - will only be propagated
[13] The API supports the operations, however the existing implementation to use reasoning provide only partial support. Operation are supported only in single user mode (if parallelism is blocked and all requests are handled in a simple sequence).

	out	rd	rd without Space	rdmultiple	subscribe	unsubscribe
DistributionManager	Y	Y[8]	Y[8]	N[9]	X	X
MetadataManager	Y	Y	Y	Y[11]	Y	X
SecurityManager	Y	Y	N[12]	Y	X	X
TS Adapter	Y	Y	Y	Y	Y	Y
TS API	Y	Y	Y	Y	Y	Y
QueryProcessor	N	Y	N	N	N	N

Table 4.2: Supported functionality of the extended API.

	in	inmultiple	create	get	begin Transaction	commit Transaction	rollback Transaction
DistributionManager	X	X	X	X	X	X	X
MetadataManager	X[11]	X[11]	X	X	X	X	X
SecurityManager	X	X	X	X	X	X	X
TS Adapter	Y	Y	Y[13]	N	Y[13]	Y[13]	Y[13]
TS API	Y	Y	Y	Y	Y	Y	Y
QueryProcessor	N	N	N	N	N	N	N

Table 4.3: Supported functionality of the further extended API.

The blue line indicates the read and the orange line the write transactions. The red line depicts the number of modified files. A fast increase of activity can be observed since the implemenation start in November 2007 which peaks in March 2008. The implementation improved continously in the last months. This is also reflected in the curve since the number of subversion transaction continously increases. The regression in December and January can be ascribed to the Christmas holidays.

Date	Read Transactions	Write Transactions	Total Files Updated
Mar 2008	633	223	1,240
Feb 2008	926	143	844
Jan 2008	348	101	809
Dec 2007	337	152	1,019
Nov 2007	19	14	86

Table 4.4: Subversion actity since November 2007

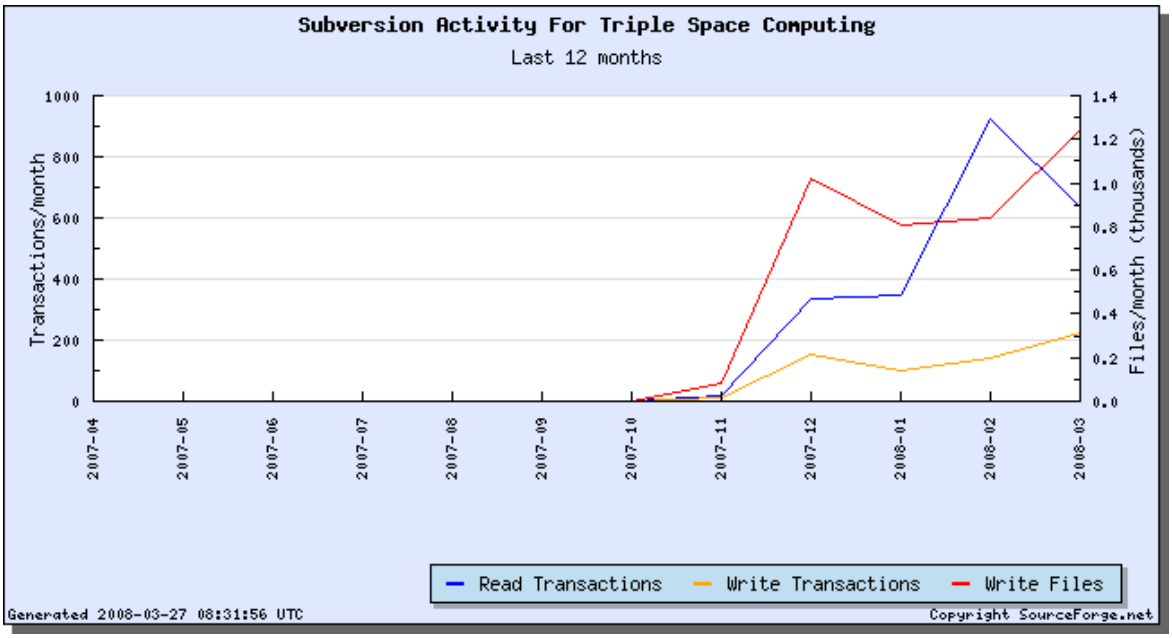

Figure 4.2: Subversion activities in the last 12 months

5 Conclusion

In this deliverable we have presented the work done to realise the first TripCom implementation. A major part of this was to integrate the separate components of the Triple Space kernel. We chose to do this using the tuplespace communication paradigm, achieving a loose coupling of components. This architecture has many benefits, as kernels could be parallelized, with multiple instances of a component sharing the same integration bus to allow for greater concurrency in Triple Space accesses. The system bus, as a space, can also be split into spaces to allow for components to work in a more efficient co-ordinated fashion. We have successfully integrated the first versions of our components in a Triple Space kernel as well as enabled the distribution of Triple Space over multiple kernels which connect to one another in a Wide Area Network. Other deliverables describe in more detail the functionality and evaluation of individual components, while a special deliverable on Scalability will consider current performance factors in the first TripCom implementation.

Our research will continue in the remainder of the TripCom project to develop second versions of individual components which provide extended or improved functionality, and their integration into a second TripCom implementation will be described in a subsequent deliverable and act as the major output of the project. Already, the first implementation is available on the Sourceforge site. In particular, we focus on more complex co-ordination patterns beyond those enabled by the core API methods as well as improved distribution and discovery of triples in a Web scale Triple Space. Hence our first implementation serves as a testbed and development platform for the final Triple Space platform we are developing in TripCom, aiming both at the challenges of Web scale persistent publication of semantic information as well as realising through that functionality two exemplary scenarios: Europe-wide Patient Summaries and a digital marketplace for media assets.

References

[1] Germán Toro del Valle, Henar Muñoz Frutos, Daniel Wutke, Dario Cerizza, and Jacek Kopecky. Methodology for adopting Semantic Web Services in a Triple Space environment. TripCom Deliverable D4.3, April 2008.

[2] Tony Andrews et al. Business Process Execution Language for Web Services, version 1.1, July 2002. Available at: http://www.ibm.com/developerworks/library/ws-bpel.

[3] Philippe Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.

[4] R. Krummenacher, E. Simperl, V. Momtchev, L.J.B. Nixon, and O. Shafiq. Specification of the Triple Space Ontology. TripCom Deliverable D2.2, March 2007.

[5] Vassil Momtchev, Stijn Heymans, Jos de Bruijn, Axel Polleres, and Lyndon Nixon. Triple Space Knowledge Representation. TripCom Deliverable D2.3, March 2008.

[6] Martin Murth, Gerson Joskowicz, eva Kuhn, Dario Cerizza, Davide Cerri, David de Francisco, Alessandro Ghioni, Reto Krummenacher, Daniel Martin, Lyndon Nixon, Nuria Sanchez, Brahmananda Sapkota, Omair Shafiq, and Daniel Wutke. Triple Space Reference Architecture. Technical report, TripCom, FP6 - 027324, 2007.

[7] Lyndon Nixon, Elena Paslaru Bontas Simperl, Reto Krummenacher, Francisco Martin-Recuerdaa, Martin Murth, Geri Joskowicz, and eva Kuhn. Specification and implementation of a semantic linda model. TripCom Deliverable D3.1, 2007.