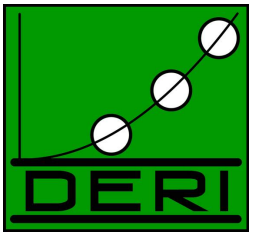
University of Innsbruck
Digital Enterprise Research Institute



Java API for Mobile TSC/YARS

Bachelor Thesis

Matthias Farwick
Daxgasse 11
A-6020 Innsbruck
Inscription No.: 0318424

Supervised by Reto Krummenacher

Innsbruck, July 2006

Abstract

It is a widely accepted fact that the asynchronous publish and read paradigm is a major reason for the successfulness of the World Wide Web. The research field of Triple Space Computing(TSC) tries to create such an infrastructure, not for human consumers, but for asynchronous process to process communication. Asynchronous communication is especially important for small and mobile devices, which cannot always be online. This thesis introduces the current research status on TSC, and uses this knowledge to implement a Triple Space architecture, which is able to store and retrieve semantic data to and from a data store. It also provides two TSC API implementations enabling developers to conveniently create mobile and desktop applications that interact with the space.

Contents

1 Introduction

Triple space computing is a new paradigm for process to process communication. This research field tries to create a scalable middleware for semantic information for machines, much alike the World Wide Web for humans, with special emphasis on Semantic Web services and ubiquitous computing. The development of this technology is pushed by DERI Innsbruck, the Vienna University of Technology, the National University of Ireland, Galway and the two companies Electronic Web Service GmbH, Innsbruck and Thonhauser Data Engineering GmbH, Leoben(Steiermark). It was founded by the FIT-IT research programme in the programme line of "semantic systems and services".

TSC is based on the fusion of two main technologies: tuple space computing [7] and Semantic Web technologies in particular RDF [12]. These technologies should provide an environment which decouples the three dimensions of process communication: reference, time and space. In a triple space, information is stored as triples [12] in a shared space, which can be queried and altered by all connected devices, i.e. desktops, laptops, sensors and even mobile devices.

If, as example for time autonomy, a desktop PC wants to send a message to a mobile device, it puts the message into the shared triple space. When the mobile device then establishes a connection to the space it can retrieve this message.

The time autonomy especially benefits this integration of mobile devices into such an environment, because those devices often have an unstable network connection and are therefore not always available.

When advertising and applying new software technologies, which are unknown by the economy and potential sponsors, reference implementations, which show the capabilities and potentials of a fresh technology, are of great importance. Since one of the major fields where TSC can play out its advantages is ubiquitous computing, the focus is laid on this area of work in this bachelor thesis. Ubiquitous computing tries to embed computation into every situation of the everyday life of a person. The workshop paper "Sharing Context Information in Semantic Spaces" [15] shows how ubiquitous computing could come into play in this context.

To develop such a reference implementation, application programmers for mobile devices need a simple API to conveniently interact with a triple space implementation. The main aim of this bachelor thesis will be the discussion of an API for mobile devices, and a reference implementation written in J2ME [19]. To test this implemented API the implementation of a reference space is also needed.

The remainder of this thesis will be structured as follows. In Section 1.1 the goals of this thesis will be laid out more precisely and Section 1.2 describes the methodology of how this goals will be achieved. The overall architecture and the basic concepts of Triple Space Computing will be discussed in Section 2. In this section also the proposed API, which was a model for the implementation, is introduced. Section 3 introduces the background technologies involved in the implementations made for this thesis. The two API implementations for this thesis will be described in Section 4. These are a J2SE and a J2ME implementation of the API. Of course, these API implementations need a space implementation to communicate with. This reference implementation of the space is described in Section 5. After describing all reference implementations, Section 5.4 shows an example of a client - space communication using the reference space, and API implementations. Section 6 analysis the differences of the reference implementations and the proposed TSC architecture of the Triple Space Computing project consortium. Finally Section 7 concludes this thesis and gives some final thoughts about this interesting research field.

1.1 Goals

The main goal of this bachelor thesis is the development of an API, written in Java, which enables developers of software for mobile devices, to conveniently interact with a space. The methods outlined in [16] Table 2.1 will form the basis for this API, but will be discussed, altered or even discarded. The API implementation should handle the RDF data and the queries as Java classes, which represent the data in an easy to handle way. It is the objective to continuously test these methods during the development, to make their behavior stable and deterministic. Another goal of this thesis is to create a space implementation, which enables testing of the API, and provides some basic functionalities of a space. One of these functionalities is for example the creation of a globally unique identifier for inserted data.

This thesis shall also give an introduction to the current state of the art in Triple Space Computing, revealing the advantages and the potentials of these technologies in the future.

Another goal, is to provide a documentation of the development of the API, and also how it is supposed to be used.

Developing a reference client, which utilises this newly developed API, will be the topic of a parallel bachelor thesis by Mark Mattern [18].

1.2 Methodology

There already exist a lot of ideas about how a TSC API should look like and what it is supposed to be able to do [16] [4]. As a starting point the core API from the TSC Project Deliverable 1.2 [16] Table 2.1 will be taken, but it will discussed and maybe even altered. Before implementing the API, many J2ME specific topics have to be thought through. For example, should threads be used to guarantee usability during lengthy connection operations? Or more fundamentally, where lie the differences between J2SE and J2ME programming. It also has to be determined, if there already exist classes which represent RDF data, and if these can be used in an J2ME environment. Furthermore it is important to initiate a good communication to the developer of YARS. This is important to be able to react on changes made on the YARS storage system. When these basic questions are answered the development of the reference implementation of a basic space implementation can begin. From there on a J2SE implementation of the API should be developed to get a feel for how the interaction between a client, the space and the storage system(YARS) works. During this process classes which handle RDF information should be created which do not contradict with the mobile programming environment J2ME. When these classes are created and proofed to work in a mobile environment, the implementation of the J2ME API can begin. Of course all these classes and API methods will be continuously tested with small Java test clients, to ensure the right functionality of the methods. During all stages of development document the creation process will be documented.

In this section the methodology and the goals of this thesis have been laid out. The following section will show the state of the art of the overall architecture of the Triple Space concept. This will give the reader the possibility to understand how the reference implementations, which are described in the Sections 4 and 5, are set up and how they work.

2 The TSC Concept

Triple space computing is a new paradigm for process to process communication. This research field tries to create an infrastructure of semantic information for machines, much alike the world wide web for humans.

It especially aims on Semantic Web Services. The current communication model of Web Services is based on synchronous and statefull message exchange, which is in contradiction with the Web paradigm of persistent pub-

lish and read, also referred to as the REST principle [6]. This means that in the traditional setup Web Services and clients need to be online at the same time, share a common data representation and know each other. Also the exchanged messages contain the actual data to be processed, and do not refer to it as resources on the web, using URIs [8]. This is also a violation of the REST principle.

Similar to tuples spaces introduced in [7], Triple Space Computing is supposed to provide a shared memory called Triple Space. The Triple Space provides an environment for asynchronous communication of Semantic Web Services, but also other processes. In this model a Web Service publishes a message on the Triple Space. Another Web Service can then fetch this message from the space, work with it and then republish the result. This mechanism behaves just like the World Wide Web for humans, where humans publish information on Websites which can later be retrieved by other humans.
An architecture of this kind has several advantages over the traditional communication model. These are:

- **Space autonomy:** meaning that two services can run in completely different environments, as long as both can access a given Triple Space.
- **Reference autonomy:** meaning that communicating services do not need to know each other, because the communication runs over the shared space. This means that Web Services can provide functionality without ever having to directly communicate with another peer then the space.
- **Time autonomy:** meaning that two communicating services do not need to be online or running at the same time. This is especially interesting for the inclusion of mobile devices which do not have guaranteed uptime. For example a service publishes a request on a Triple Space that can only be processed by a service running on a mobile device i.e. a PDA, which is not online in that moment in time. When this mobile device finally goes online it can retrieve and handle the request from the space and then publish the result.
- **Semantic autonomy:** meaning that services which do not share the same data representation format can communicate with each other. This can be achieved through mediation processes implemented in the space.

The following subsection will show how a space can be located in a global network and how high scalability can be achieved through so called subspaces.

2.1 The Triple Space

A Triple Space must be accessible and localizeable in a network. In most cases this network will be the Internet but also local Triple Spaces can be implemented. To identify a Triple Space and to finally communicate with it, every space needs a globally unique identifier. This is a URI [8] just like every Website has a URI. A URI is composed of a protocol (for example http), an authority (for example example.org) and a path leading to a specific resource in the authority. Through such an URI a space could be uniquely identified. But this does not mean that a space is situated on only one certain server. A space could be mirrored to many different servers, meaning that a change made in one space location would have to be also made at the other space locations. This mirroring will especially be important for highly used spaces, where the traffic load can be spread among several machines. Section 6.1 will show which components of a Triple Space implementation are involved in this process.
Another important scalability aspect of Triple Spaces are so called Virtual Subspaces. This idea was introduced in [13]. It is supposed to logically separate information, and also to provide restriction for different user groups to access certain information in the space. The following two URIs could describe two different subspaces intended for two different user groups. The first subspace is supposed to be read by everyone, the latter only by DERI members.

```
http://www.example.org/space/public_space
http://www.example.org/space/private_space
```

Although both subspaces store the data in the same space, the data is tagged, so that on retrieval it can be checked to which subspace it belongs.

The TSC project Deliverable 1.3 [13] proposes in allusion to [3] that the scheme part of the URI of a Triple Space should be independent of underlying implementation. Because of this the scheme part in the examples above (http), would be replaced with *ts* for Triple Space. In an application, this scheme would be then mapped to the appropriate underlying space communication protocol. This would lead to the following Triple Space URI notation:

```
ts://www.example.org/space
```

While this subsection described how a Triple Space can be addressed in a global network and how data can be logically separated in a space, the next section will deal with the data model in which data will be stored in a space.

2.2 The TSC Data Model

Triple Space Computing focuses on storing semantic data. How this data is arranged and stored in a space implementation will be described in this section.
RDF [12] is the main data representation language in Triple Space Computing. More information on this topic can be found in Section 3.1. In Triple Space Computing data is stored in the form of triples. These triples are connected with an ID to identify a set of triples in a certain space. Together with the URI of the space this ID creates a unique identifier of the data on a global network. This identifier is called the URI of the triple or triples. A triple connected with an identifier is called a Quad. More on Quads can be found in Section 3.1. Having a URI for a triple or a set of triples means, that triples could be simply retrieved by pointing a web browser to:

```
ts://www.example.org/space/public/triples_about_matthias_farwick
```

Of course assuming that this part of the space is publicly available. In a real implementation the part *triples_about_matthias_farwick*, would not be such an expressive string. How this part could be generated by the space and kept unique will be explained in Section 5.2.

The triples identified by the URI are called a "Named Graph". It has to be mentioned that in the context of TSC a graph is distinct from a named graph. A named graph is a graph which can be identified by an URI. A simple graph is not connected to such an identifier. More on this topic can be found in [11].

Deliverable Deliverable 1.3 [13] also explains that it is rational not to identify a single triple by an ID but rather a whole graph. This way entire objects can be addressed, but also single triples when the graph consists of only one triple. Using the API function *write* explained in Section 4 to write a whole graph describing a person will result in only one URI of the graph. Since the graph can now be identified with an URI it is now a named graph.

```
Vector personTriples;
...
/*
 * For example the vector contains many triples,
 * using the foaf ontology to describe a person.
 */

URI spaceURI = new URI("ts://www.deri.at/tsc/space");

URI id = write(spaceURI, personTriples);
```

An application which wants to retrieve all information about a certain person could use the URI produced by the *write* function to retrieve all triples about this person.Of course the question arises, who generates the ID of a written graph and how it is generated. Concerning the first question an ID has to be unique and valid. Because of this, it is obvious that not a potential client of a space decides which ID to choose. This is due to the fact that only the space can know which IDs are already assigned and which ID format is chosen by the space implementation. The second question, how the ID is supposed to be composed, is an implementation specific detail. One solution would be to generate a hash value over the data, the authors URI and the timestamp of a written graph. How this problem was tackled for the reference implementation will be explained in Section 5.2.

As described in Deliverable 1.4 [20] the minimal requirements for trust to triples, are a timestamp and the author of the triple. For this reason a space has to store this information, for all named graphs in the space. This enables any application using a named graph, to consider whether it wants to trust the correctness of a given triple or not. More information concerning security and trust issues in Triple Space Computing can be found in [20].

This section described the data model of Triple Space Computing, that allows unique identification of data, through the usage of named graphs. In the following subsection the overall architecture of the proposed technology will be described. This will provide a brief overview of the current state of research. Through this the reader will be enabled to see which of these parts were implemented for this thesis.

2.3 The Proposed Triple Space Architecture

The current research on Triple Space Computing foresees a stacked layer architecture of a Triple Space. The proposed architecture is explained in detail

in the TSC project deliverable 2.1 [4]. A brief overview of the proposed components will be presented here, and in Section 6.1 where the the implemented architecture will be compared to the proposed framework. A key concept of the current research, is the concept of communicating spaces which share the same data and inform the other spaces upon changes. This communication can be seen at the right side of the image where the coordination layer handles the synchronisation of coupled spaces. This concept is very important to guarantee the scalability which is needed to provide global access to a space with heavy usage.

The layers of the stack architecture should abstract from underlying layers. Lower laying layers for example are responsible for low level operations, and higher level stacks can use this functionality without know how it is done. The top layer in such in architecture is always responsible for interaction with some forms of clients.

Clients

The TSC architecture foresees two kinds of clients.

1. **Light Clients:** these clients are the traditional form of clients, meaning that they are only able to request or enter data into the space. A possible way of communication between client and space is HTTP. Messages to and from the space are handled via a servlet [22] implemented as the interface of the space. Another way would be light clients which implement a space proxy. This is the form of a client how it was implemented for this thesis.

2. **Heavy Clients:** these clients have much in common with both a space and a light client. On one hand they can take part in storing and managing data, and on the other hand they are not always connected and can even be run on mobile devices.

As mentioned earlier, the proposed Triple Space architecture does not foresee a traditional client-server architecture. Rather a super-peer approach is supposed to be taken. As explained in [26] the load of one space is supposed to be balanced on, many so called super peers, where one super peer acts as the server for a subset of clients. Having learned from traditional peer to peer systems, that slow peers in terms of connection and cpu speed, can delay a system dramatically, super-peers are mostly well connected fast machines. This enhances fault tolerance and scalability of the overall system.

A very important aspect for this thesis is, that any of these clients needs to have an interface to communicate with spaces. This important interface is the TSC API.

TSC API

The TSC API is the definition of the operations which can be made on any TSC conform space. Any space implementation must provide the functionality defined in the API. Because the proposed API is very rich in functionality, only the most significant operations will be explained here. For more information about the current research refer to [4].

Since the proposed space implementation is supposed to support transactions, all operations that involve reading and writing to the space receive a transaction parameter. This parameter specifies the transaction configuration for a certain operation. Also operations that involve querying, receive a parameter *Template*. This template specifies a certain query. In the following the notation of the TSC project deliverable 2.1 [4] is used.

```
write(URI spaceURI, Transaction tx, Graph g):
URI namedGraphURI;
```

The write operation is one of the very basic functionalities of a space. It allows to write a graph into a space identified by the space URI.

```
query(URI spaceURI, Transaction tx, Template t):
Graph g;
```

The query operation is also a very basic operation. It queries a space identified by the space URI with a given template. On success it returns a graph containing the RDF data matching the template.

```
waitToQuery(URI spaceURI, Transaction tx, Template t,
TimeOut timeOut): Graph g;
```

This operation works just like the *query* operation, but it does not immediately return if no triples were found. It requeries the space until matching triples are inserted or the timeout occurred.

```
read(URI spaceURI, Transaction tx, URI namedGraphURI):
NamedGraph g;
```

The read operation is also a basic operation. It returns the named graph identified by the URI namedGraphURI. Analogical to the *waitToQuery* operation, there also exists a *waitToRead* operation.

```
take(URI spaceURI, Transaction tx, Template t): Graph g;
```

The take operation works just like the query operation, but after returning the triples from the space it deletes them from the space storage. Analogical to the *waitToQuery* operation, also exists a *waitToTake* operation.

```
update(URI ts, Transaction tx, NamedGraph ng):
boolean;
```

This operation updates a triples identified by the template. It follows the semantics of *take* and *write* in that order.

```
subscribe(URI spaceURI, Template t, URI callBackReceiver):
void;
```

Through this operation a client can inform the space that it wants to be informed if triples are inserted or altered in the space, which match the given template. The callbackReceiverURI can point to a remote method or any other means of calling the subscriber. More on this mechanism can be found in the TSC project deliverable 1.2 [16]. There is also supposed to exist an unsubscribe operation, which removes a certain subscription.

```
notify(URI spaceURI, URI subscriptionURI, URI namedGraphURI):
void;
```

This operation is responsible for notifying any subscriber which subscribed for a certain template. The namedGraphURI points to the named graph which matches the template.

```
advertise(URI spaceURI, Template t): void;
```

Through this operation a producer of graphs can state that it intents to produce graphs that match the template t. To remove this advertisement there there is supposed to exist an operation *unadvertise*.

```
count(URI spaceURI, Transaction tx, Template t): long;
```

This operation returns the number of triples which match a given template in a certain space. This could be used to estimate the amount of bytes which have to be transferred to the receiver, in case a very large number of triples is expected to match.

While this part introduced the proposed TSC API, Section 4, will discuss the implemented TSC API for this thesis.

This part described the overall architecture of a Triple Space environment. But what are the goals of Triple Space Computing for the future? The next part tries a brief outlook.

2.4 Future Goals of TSC

As mentioned earlier Triple Space Computing has the capability to overcome many of the problems arising through the heterogeneity of devices connected to the web. Thus it is the major goal of TSC to become the "Web for Machines", as the WWW is the web for humans (see [14]). It's portability, flexibility and the already mentioned ability of meditation between data formats make it especially interesting for ubiquitous computing. Also its ability to provide asynchronous message exchange emphasis this. The workshop paper "Triple Spaces for a Ubiquitous Web of Services" [17] gives an overview on the advantages of TSC for ubiquitous computing.

This section introduced the general concepts of Triple Space Computing, as well of the future goals of this proposed technology. The next section will give a brief introduction to the background technologies involved in the implementations for this thesis.

3 Background Technologies

In the previous section the proposed architecture was introduced. Since this thesis is about an implementation of a Triple Space and its mobile interface, some technologies which are essential for such an implementation have to be introduced. The following subsection 3.1 will introduce RDF as the major form of storing and handling data in TSC. Subsection 3.2 will describe YARS, the RDF storage facility which was used for this implementation. Finally subsection 3.3 will conclude this section by introducing J2ME, as the environment for programming mobile devices.

3.1 RDF - N3 - Quads - N3QL

In the context of Triple Space Computing RDF [12] is the main format of data representation. RDF is a specification for metadata, implemented in XML [5] or other languages. It stores data in so called triples which consist of a subject, a predicate and an object. Where the subject is the "thing" which has to be described, the object is the value which is brought in relation with the subject. And finally the predicate describes the kind of relation between subject an object.

In the context of this thesis, triples were modeled as java classes to be able to conveniently handle the RDF data. The explanation of this implementation can be found in appendix B.

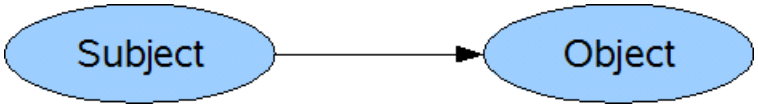


Figure 1: Semantics of a RDF triple

In the case of YARS and the reference implementations of this thesis, N3 [1] is used to represent RDF data. This is a notation developed by Tim Berners-Lee, which has a focus on better human readability than RDF/XML and also has a smaller textual overhead. For example the repetition of another object for the same subject and predicate can be achieved by only using a comma "," separating the different objects.

More information about N3 can be found on the official specification Website of the W3 at [1].

Quads

While it is possible to make complex statements with the help of RDF, it is very costly to make statements about other RDF statements. Due to this lack of flexibility quads are introduced to provide an identifier for triples, so triples can be described. In regard to quads, a triple then consists of not only a subject, a predicate and an object, but also of an unique identifier(ID). Using the unique ID as a subject of a RDF statement, one can make statements about statements, and identify a triple in a certain scope. In the context of this thesis the named graph is moreover used, to refer to a triple or a set of triple with the same ID.

More details about the use of quads in TSC and the composition of the unique IDs can be found in Deliverable 1.3 [13].

N3QL

N3QL is a querying language for RDF data. It lets the programmer describe the belonged RDF data in form of triple constraints and patterns. It also specifies the format of the returned data. In the case of this thesis, this querying language is used to retrieve RDF data in the N3 format from the data store YARS (see Section 3.2). More information about this querying language is available at [25].

3.2 YARS

Any triple space needs a secure, fast and reliable persistency framework to store the semantic data. This is especially important because Triple Spaces

might handle orders from online shops etc. which require special security needs. There exist several implementations that are built to store RDF data, but most of those have a poor querying performance as it can be seen in [10]. In contrast, a lightweight persistence framework is developed in Java at DERI Galway, which uses optimized indexes for better querying performance. For this reason, and also because DERI Galway cooperates with DERI Innsbruck, YARS has been chosen to be used for reference implementations of Triple Space Computing. YARS also has the major advantage of being able to store quads instead of RDF triples. This allows for usage of the already mentioned named graphs.

YARS works as a servlet [22] which can be stored and run on a web container like Tomcat [24]. The servlet takes requests and manipulates the underlying database implementation. This implementation is the Berkeley DB, a pure Java application, which is lightweight and run from a single jar-file. More information about the underlying database can be found here [21]. Actions on the YARS storage system can be made, like with all servlets, through simple HTTP GET,PUT and DELETE requests. With these three request the basic database actions querying, inserting and deleting can be performed.

When performing a GET request to query YARS, a client has to append *?q=N3QLqueyString* to the YARS URI. YARS then extract the query string from the URI, and queries its database for the requested triples. It then returns them within a HTTP response with the response code 200 for "OK". If the query was malformed or no triples for this query were found, YARS returns the response codes 400 for "Bad Request", or 204 "No Content" respectively. A client can also specify the ID of a certain graph, the client wants to retrieve. This is done through attaching the ID to the URI of YARS, and then performing an HTTP GET request on this URI. Additionally a query can be used as seen above to only return parts of the named graph identified by a certain ID.

The HTTP PUT request is used to insert triples into YARS. The client, in the case of TSC the space, can specify an ID to afterward identify the written triples. Inserting works as follows.

Assumed that YARS is runs at *http://www.deri.org/yars*. If a PUT request is performed on *http://www.deri.org/yars/ID-specified-by-requester*, the triple will be stored with the ID "uri-specified-by-requester". This way a triple or a named graph is identifiable through its the URI that is composed by the YARS URI and the ID specified by the requester. When the insertion was successful HTTP servlet response with the code 201 for "Created" is returned to the requester. When errors occurred either 204 for "Bad request", or 500 for "Internal Server Error" are returned.

Deleting works in a similar way like the GET request but instead of returning the triples matching a query, these triple are removed from YARS.

During the work on this thesis it was possible to propose and apply several enhancements on the implementation of YARS. For example there was no content type defined for any returned triple. This had to be changed because it is a necessity for a storage system to declare the content type of any returned data. In the TSC world for example this data type could be RDF, N3 or other languages to describe semantic data. This necessity was proposed to the developer of YARS Andreas Hardt and then implemented.

Furthermore there exists a YARS API. This API takes the burden of dealing with HTTP requests to communicate with YARS from a developer creating clients for YARS. The API was inspired by the Sun's JDBC API, and therefore the usage is very similar. The usage of the API is explained in Appendix C.

The API also contains classes representing RDF triples and a parser for N3. The parser is able to take an N3 string and convert it to Triple classes representing the N3 data. Resources and Literals themselves are also represented as classes. All these classes implement a *toN3()* function which returns the N3 representation of the object. More on the implementation of the triple classes can be found in Appendix B.

3.3 J2ME

What people commonly refer to as "Java" is either the Java 2 Enterprise Edition (J2EE) or the Java 2 Standard Edition (J2SE). But there exists a third edition called Java 2 Micro Edition(J2ME). Each of these editions has its special environment for which they were developed. J2EE is made for server applications, J2SE for desktop applications and finally J2ME is made for small limited devices such as mobile phones or PDAs. All these editions have their own Virtual Machine (VM) and their own set of runtime classes. To be precise, the runtime library of J2SE is a subset of the library of J2EE and the J2ME library is a subset of the J2SE classes. This corresponds to the memory and processor speeds of the given devices. It is possible to write classes which run on all three environments but it has to be said, that the library of J2ME is very limited due to the storing capabilities of most small devices. Like for J2SE and J2EE, the specifications that comprise J2ME are developed by the Java Community Process (JCP) and can be found at [23].

J2ME is made to adapt to limited devices. These limitations are small

memory, small screen sizes, limited input capabilities and slow processors. Of course, each device differs in these limitations. To adapt to the given circumstances of different devices the J2ME developer has to arrange three different core concepts. These are configurations, profiles and optional packages. Those concepts will be described in the following.

Configurations

A Configuration is a complete Java Runtime Environment consisting of

- A Java virtual machine (VM) to execute Java bytecode.
- Native code to interface to the underlying system.
- A set of core Java runtime classes.

To use a certain configuration a device has to comply to minimum requirements. There are two different Configurations. One being the *Connected Device Configuration* (CDC) and the *Connected Limited Device Configuration* (CLDC). CDC requires much more memory and processor speed than the CLDC, which means that the Runtime library of the CDC is bigger. In fact the library of the CLDC is a subset of the CDC. Both implement only a fraction of the core classes of the J2SE Runtime Library, from the namespaces *java.lang*, *java.io* and *java.util*. The classes contained in a configuration do not contain classes to implement graphical user interfaces. These classes are mostly added by the chosen profile explained in the next part.

Profiles

Profiles are even more environment specific. They add more classes to the available library and mostly provide classes to support interactive applications. Several Profiles exist, for example the *Mobile Information Device Profile*(MIDP), which was developed for mobile phones, and provides wireless HTTP connectivity. This was especially important for this thesis. Another Profile is the *Personal Digital Assistant Profile*(PDAP) for PDAs. It extends the MIDP, with more classes, because PDAs in general have more memory and bigger screens than mobile phones.

Optional packages

Optional Packages are there to add specific functionalities like Bluetooth support. They have their own requirements to their environment, and their own dependencies on certain profiles and configurations.

After seeing which options a J2ME developer has, the the question arises what a J2ME application really is. This question will be discussed in the next part.

Problems

From the section above it should be apparent that the term "J2ME application", cannot describe an application really well. It is also clear that when a developer decides on Configurations, Profiles, and Optional Packages, he restricts the possible devices which can later run the application. For example if a developer chooses to use the CDC, he can program with more comfort because he can access a bigger library, than when using the CLDC. But the developer also leaves out most mobile phones and other small devices, to be able to run the application. On the other hand, if one chooses to use the CLDC, one can program with much less comfort. Thus it is important to carefully choose to overall configuration of an J2ME application prior to the start of any coding.

Which setup was chosen for the mobile API developed for this thesis will be explained in Section 4.

This section introduced the basic technologies involved in Triple Space Computing, and also the technologies which are important for mobile computing. The following section will focus on the API which is needed to communicate with the space, the TSC API. Also the J2ME and the J2SE implementation for this thesis will be described.

4 The TSC API Reference Implementations

The TSC API is the main communication interface of any Triple Space implementation. Much effort has been put in the design of this API. The more complex a space implementation gets, the more defined functions are needed to interact with the space. If one reduces the complexity of a space implementation, only a couple of basic functions are left. In the context of this thesis only these very basic functions were considered. This allows for a basic proof of concept. As already seen in Section 2.3 the current proposal for the TSC API contains many functionalities for many different purposes.

This section focuses on the two different reference API implementations and their design. One being implemented in J2SE for desktop applications, and

the other written in J2ME to be used for mobile applications. Both share the same interface but differ radically in their implementation. The basis for design of the interface was formed by the TS-Core API and the TSC API with N3QL query support, both proposed in [16]. Table 1 shows the API operations proposed in [16] compared to the implemented operations. This table will be explained below.

TSC project deliverable D1.2	Reference impl.
1: write(java.net.URI ts, Triple triple): URI	write(I_URI ts, I_Triple triple):I_URI (java.net.URI not available in J2ME)
2: write(java.net.URI ts, List<Triple> triples): List<URI>	write(I_URI ts, Vector triples):Vector (List/URI not available in J2ME)
3: read(java.net.URI ts, java.net.URI triple): Triple	read(I_URI ts, I_URI triple):I_Triple (java.net.URI not available in J2ME)
4: read(java.net.URI ts, Set<URI> triples): Set<Triple>	read(I_URI ts, Vector triples):Vector (Set/URI not available in J2ME)
5: take(java.net.URI ts, java.net.URI triple): Triple	not implemented(persistent publishing)
6: take(java.net.URI ts, Set<URI> triples): Set<Triple>	not implemented(persistent publishing)
7: take(java.net.URI ts, String query): Set<Triple>	not implemented(persistent publishing)
8: waitToTake(java.net.URI ts, String query): Set<Triple>	not implemented(persistent publishing)
9: waitToTake(java.net.URI ts, Set<URI> triples, long timeout): Set<Triple>	not implemented(persistent publishing)
10: waitToRead(java.net.URI ts, URI triple, long timeout): Triple	not implemented(error in semantics)
11: waitToRead(java.net.URI ts, Set<URI> triples, long timeout): Set<Triple>	not implemented(error in semantics)
12: waitToTake(java.net.URI ts, java.net.URI triple, long timeout): Set<Triple>	not implemented(error in semantics)
13: read(java.net.URI ts, String query): Set<Triple>	read(I_URI ts, String query):Vector (Set/URI not available in J2ME)
14: waitToRead(java.net.URI ts, String query): Set<Triple>	waitToRead(I_URI ts, String query):Vector (Set/URI not available in J2ME)
15: count(URI ts, String query): long	count(I_URI ts, String query):long (java.net.URI not available in J2ME)

Table 1: The proposed API of [16] and the reference implementation.

During the discussion and implementation of the API some functions were discarded from the proposed TSC API by the research group, and some were discarded and later added again. The current result of this evolvement of

the API is the API definition shown in Section 2.3. At a certain stage of the implementation part of this thesis, it was not reasonable anymore to adapt to the changes of the TSC API definitions made by the TSC research group. This explains why some operations which are currently proposed for the TSC API were not implemented for the reference implementations. Some operations were also out of the scope of this thesis. Table 1 shows which operations from the given API definition were implemented, for which reason some were not implemented, and how others were altered. In the following the changes which were made in the API are discussed, starting with the *write* operation at the top.

In the first line of Table 1, which contains a *write* operation, it can be seen, that the URI class was not used in the implemented version. This is due to the fact that the used CLDC(see Section 3.3 for the J2ME implementation does not contain the java.net.URI class. Therefore an interface and J2ME and J2SE specific implementations for the *I_URI* class were developed. One can also see that not a *Triple* class is used, but an *I_Triple* interface is required in the implemented API. Because the J2ME and the J2SE implementations had to use two different implementations of the *Triple* class, which implement the same interface, this architecture was chosen. More on these topics can be found in the implementation specific Sections 4.1 and 4.2. This example, and others, show how the design of the reference API was influenced by the requirements of the used J2ME configuration. The same applies for the write operation of line two. Here the API blueprint demanded a *List* object as the input of the function. But the class is also not existent in the used J2ME environment. As a solution the *Vector* class was chosen, which has similar behavior, and is available in both J2ME and J2SE. Here it can also be seen that the API specification of [16] demanded a *List* object filled with Triple objects. The concept of generic collections is also not supported in J2ME. Therefore this feature was left out for the reference implementation. The rows three and four show similar changes as described above. The *take* operations of the lines 5,6,7,8 and 9 are examples for functions which were proposed in [16], discarded by the research community and later added again to the TSC API. During the time of the implementation of the reference TSC API, the functions were removed from the proposed TSC API. This was due to the opinion of the TSC research group, that only if data cannot be deleted from the space, it fully complies with the REST principle described in Section 2. It was the believe of parts of the research community that, through a timestamp the latest information can always be found in a space [14]. This would remove the need of a delete/update/take functionality and also enable rollbacks to previous statuses of a space. This was not the opinion of all re-

searchers(and the author of this thesis), because also problems can arise due to the immense quantity of data that would accumulate in a space. If data would never be deleted or updated, the query performance would also sink. It can be seen in Section 2.3 that for the current version of the TSC API in [13], the researchers finally stepped back from the persistent publishing and added take and update operations to the proposed API again.

During the discussion and the design of the reference API implementations, it was possible to discover mistakes and alter the proposed API. Some mistakes were found in the blocking operations of the lines 10,11 and 12 of Table 1. These operations have an error in their semantics, because they wait for triples to be saved in the space. The problem is that the invoker of these functions cannot know the URI of the triples he wants to search for, prior to the writing them. This means that these operations will never fulfill their function of waiting for triples which do not yet exist. They either return the triples immediately if they exist, or they will block until the timeout occurs and will return nothing.

Finally the operations of the lines 13 to 15 were implemented with the changes from *URI* to *I_URI*, *List* to *Vector* and from *Set* to *Vector* as explained above.

The following part will describe the implemented TSC API and the implementation specific details. It also explains what an applications developer has to consider when using one of these TSC API implementations.

The implemented TSC API

In the context of this thesis, two reference implementations of a TSC API were developed. One being implemented in J2SE the other being implemented in J2ME. In combination with the space implementations described in Section 5 and a YARS distribution described in Section 3.2, developers can setup a space and use the APIs to communicate with it. For the Bachelor Thesis of Mark Mattern [18] a mobile and a desktop client were implemented to show the capabilities of this setup. Figure 2 illustrates the layering used by the reference clients.

Figure 5 shows that a proxy implementation over HTTP was chosen here. When using one of the API implementations the proxies are seen as a local object by the client application. A complete interaction example, between a client using one of the APIs, the space, and the storage system YARS, is given in Section 5.4.

Both implementations establish connections to the space to commit possibly

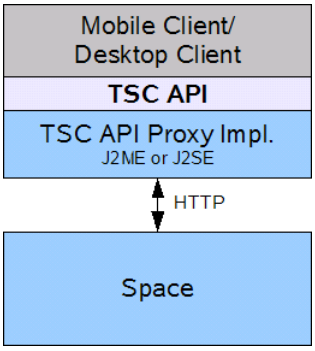


Figure 2: The client - space communication

lengthy store and retrieve operations on it. For any developer who needs all-time responsiveness of his graphical user interface, it is important to consider the use of worker threads, when using the API functions. This way it can be guaranteed that user input can still be handled, while lengthy space operations(possibly transmitting thousands of triples) are ongoing. Otherwise one of the API operations will block any input to the graphical user interface, while the operation has not returned. For a code example on how to create such a worker thread in J2SE or J2ME please refer to Appendix A. Using such a thread leaves the main thread idle to receive input.
The application developer is also responsible for catching the exceptions thrown by the API functions. The handling was deliberately left up to the client developer, because this way the developer can find out, which problem caused the exception that was thrown. Also the developer can log the exception. Since the space implementation returns response codes according to the successfulness of the operations, as described in the following Section 5, it is possible for the API implementations to generate meaningful exceptions. The following function signatures are the operations which are implemented as the TSC API in the context of this thesis. They can be considered as a snapshot of the theoretical development of the TSC API in autumn 2005. This snapshot differs from the proposed API in [16], for the reasons already explained in this section.

```
Vector read(I_URI spaceURI, String n3query)
throws IOException;
```

This API function queries the specified space. As arguments it takes a space I_URI *spaceURI* and the string *n3query* specifying a certain N3QL query pattern. If triples matching the query were found it returns a Vector filled with N3 triple objects. If no triples were found in the space it returns *null*. If

the function failed to query the space, for example if no network connection exists, the function throws an *IOException*.

```
Vector read(I_URI spaceURI, I_URI graphURI)
throws IOException;
```

This API function returns a named graph identified by an URI. As arguments this function takes a space URI *spaceURI* and a *I_URI graphURI* specifying the URI of a certain graph. If the graph with this URI exists, the function returns a Vector object containing the triple objects which form the named graph. If the belonged named graph does not exists the function returns *null*. If the function failed, an *IOException* is thrown.

```
Vector waitToRead(I_URI spaceURI, String n3query, long timout)
throws IOException;
```

This API function tries to query the specified space for the query *n3query*. If the query returns no result it retries the querying for the, in the parameter *timeout* specified, amount of milliseconds. If no triples were found in the given period, the function returns *null*. In the case of an error the function throws an *IOException*.

```
I_URI write(I_URI spaceURI, I_Triple triple)
throws IOException;
```

This API function writes single triple into the space. As arguments it takes a space URI object *spaceURI* and a triple object. The function tries to write the triple to the space, and returns a *I_URI* object denoting the triples' URI on success. If it fails to write the triple to the space it throws an *IOException*.

```
I_URI write(I_URI spaceURI, Vector triples)
throws IOException;
```

This API function writes a whole graph of triples into the space. As arguments it takes a space URI *spaceURI* and *Vector* object containing *Triple* objects, which form the graph. If the graph was successfully written into the space, this function returns an *I_URI* object denoting the URI of the written graph, so that the whole graph can be addressed. If the function failed it throws an *IOException*.

```
long count(I_URI spaceURI, String n3query)
throws IOException;
```

This API function returns the number of triples matching certain query. As arguments it takes a space URI *spaceURI* and a string *n3query* denoting the N3QL query string. If no triples were found, this function returns a 0. If an error occurred during the querying, the function throws an *IOException*.

As explained in Section 5 the reference space is implemented as a servlet. Therefore the TSC API proxy implementations have to establish a connection to the space servlet via HTTP. As we will see many precautions have to be taken when establishing such a connection. But obviously the API implementations were created to relieve the application programmer from being responsible for these lower level configurations.

To connect to the space and to tell the space what to do, one or more parameters have to be appended to the URI of the space, so the space knows which functions to execute. Also triple data is appended to the URI string.

All functions have to add the *&method=...* parameter to tell the space which function to execute. For example:

```
http://www.example.org/space&method=write
```

The space also needs to know what to write to the space. Because of this the parameter *?triples=...* has to be appended to the URI to specify the triples which are supposed to be written to the space. These triples have to be in the form of a string in the N3 format, explained in Section 3.1.

```
http://www.example.org/space&method=write?triples=n3TripleString...
```

If the function requires a query instead of triples to write, the parameter *?q=...* is used instead of the *?triples=...* parameter:

```
http://www.example.org/space&method=read?q=n3qlQueryString
```

It is important to note that a URI cannot contain white space characters. But queries in N3QL and triples in N3 always contain white spaces. Because of this, the two API implementations have different ways to convert characters which cannot be in an URI, into their Hex equivalent. This detail will be further explained in the implementation specific sections below.

As it can be seen in the API specification above, some functions like the one below, receive an object implementing the *I_Triple* interface as an input parameter.

```
I_URI tripleURI = write(I_URI spaceURI, I_Triple triple)
```

This interface defines classes to handle N3 strings as objects, which can be handled and altered more easy than a simple N3 string object. It defines that triples consist of three nodes, a subject, a predicate, and an object. Also any Triple implementation must implement the method String toN3(), which returns the N3 representation of a triple object. Nodes can either be a Literal or a Resource. These classes have to implement the Node interface, which also requires the implementing class to implement the *toN3()* function. Once the triple object is given to one of the API functions, it is converted to the corresponding N3 string and then transmitted to the space over HTTP. For the J2ME implementation of the API the class *TripleImpl* implements this *I_Triple* interface. The J2SE version of the API uses the YARS API classes to implement the *I_Triple* interface. More on the implementation can be found in Appendix B.

The basic flow of any of the implemented functions is as follows. The function takes as arguments the URI of a space and another parameter indicating the operation to execute. First the method to be used, and the data to be transferred is appended to the space URI as described above. If the data contains whitespaces, the whitespace get previously removed. Then a HTTP connection is set up, where the appropriate HTTP request is selected. Where functions that read data, use the GET request, and functions that insert data into the space, use the POST request. Why the POST request is used instead of the PUT request will be explained below. The type of expected response data is set to "application/rdf+n3". After this the connection is established, and the response code is retrieved. If the response code from the space is 200 for "OK" (for read operations), or 201 for "CREATED", the response data is read from the established input stream. If triples in N3 format are read they get parsed into *TripleImpl* objects. If an URI string is received from a write operation, an URI object is created from the URI string. These objects are then returned to the invoking application. When other response codes than 200 and 201 are received, the API function throws an appropriate exception. For example, if the response code 400 for "Bad request" was received, an IOException with the exception text "Bad request" is thrown, so the application knows that the format of the request was wrong. This could happen through malformed N3QL queries for example.

The API functions and facts described above, apply for both the J2ME and the J2SE implementation. All framework specific details will be described in the next two sections.

4.1 J2SE Implementation

While the main focus of this thesis was to implement a mobile version of the API, it was important to first develop a J2SE implementation. After the implementation of the basic frame of the space, which will be explained in Section 5, it was more convenient to first implement a TSC API in J2SE, to test the space. The development of the J2SE implementation was important, to create a desktop client, which fills/queries the space and also does complex queries to show the capabilities of the space implementation. Writing such a test client in J2SE is much more convenient in J2SE. It forms a fundamental proof of concept for this TSC setup. This desktop client was developed with Mark Mattern as part of his bachelor thesis [18].

As already mentioned above the implementation needs to remove all white spaces to form well formed URI to connect to the space. This task can be conveniently done with the static function *URLEncoder.encode(query, "utf-8")*(part of the J2SE SDK 1.4), where all whitespaces and not URI conform characters are transformed a valid URI syntax.

To parse N3 strings which are received into *TripleImpl* objects, the N3 parser in the YARS API described in Section 3.2 was used. As already mentioned above the API operations which need to handle triples need an implementation of the *I_Triple* interface. Here the classes from the YARS API were used to implement the triple interface *I_Triple*. To do so, the class *TripleImpl*, simply extends the triple implementation of the YARS API, and invokes the constructor of this superclass.

As mentioned above there was a need to create a URI interface I_URI which both the J2SE and the J2ME implementation had to implement. Because the class *java.net.URI* is final it could not be extended to implement the *I_URI* interface. Thus a simple *URIImpl* class was written, with getters and setters for the URI string.

4.2 J2ME Implementation

The reference implementation of the J2ME version of the API was the main goal of this thesis. In J2SE many classes were already implemented which were needed to implement the API. Examples are the encoding of the URI to connect to the space, which is a Standard J2SE class, and also the parsing of N3 strings using the parser shipped with YARS. But the Configuration and the Profile used for this J2ME API, neither implements the *URLEncoder* class, nor do they allow the usage of the parser written for J2SE. The latter is due to the fact, that the used configuration does not implement many of classes used in the parser. Also the triple implementation from the YARS

API could not be reused for the same reason. As already mentioned above this is also the reason why the return type of read operations was switched from *Set* to *Vector*. In particular the CLDC 1.1 Configuration and the MIDP 2.0 Profile were used to develop the API. Table 2 at the bottom of this section shows which technologies were needed to be implemented separately for the J2ME implementation. Because of the problems mentioned above, alternatives had to be found to provide the same functionality with the J2ME version.

The classes which represent N3 triples had to be implemented for J2ME, creating the *TripleImpl* class. For this one had to abandon interfaces like *Synchronizable* and others, because they could not be used in this configuration. The UML diagram for these classes can be found in Appendix B.

A parser had to be developed. Although N3 strings can be nested, can contain blank notes etc., it was known at design time that YARS, and the space return triples in a much simpler format. This format always consists of one subject, one predicate and one object. Subject and predicate are always resources and the object can be either a literal or another resource. No shortcuts are used. Because of this precondition a parser could be written which is not as complex as a parser which can parse the N3 syntax in all its facets.

Also an url encoder had to be written to replace the functionality of the *URLEncoder* class from J2SE. For this reason an open source encoder was taken from [2], which transforms URL strings with white spaces into valid URL strings. For example a single whitespace ' ' is replaced with a '+'.

Another problem was that the J2ME environment did not implement the class URI. But this class was essential for the TSC API. Therefore an interface *I_URI* was created. For the J2ME implementation it was simply implemented by creating getters and setters on an URI string.

It also has to be said that the HTTP PUT request is not available with the used configuration and profile. So instead of using the HTTP PUT request to write data to the space, HTTP POST is used. Of course this had to be changed in the servlet of the space implementation as well (see Section 5).

Finally Table 2 gives an overview which technologies were needed to be implemented in the J2ME and the J2SE implementation.

In this section the two implementations of the TSC API, have been explained. The following section will describe the architecture of the space which was implemented to communicate with this API. It will also visualize and explain a complete client, space, storage, communication example.

	J2ME	J2SE
URI	not available in J2ME, implemented *URIImpl*	available but, implemented URIImpl
TripleImpl	had to be implemented	only had to be partly implemented because of YARS API
URLEncoder	had to be implemented	already existent in J2SE
N3 Parser	had to be implemented	used YARS API N3 parser
Set/List	not available in J2ME, had to use *Vector*	used *Vector*

Table 2: Illustration which technologies had to be implemented

5 The Space Implementation

This section describes the architecture of the implemented space. Also a special focus will be put on the creation process of the unique IDs of the named graphs.

In the context of this thesis the space implementation has the following responsibilities.

1. Receive write and read commands over HTTP.
2. Create unique IDs for the triples supposed to be written and store them in the storage facility. Then return the URI of the written triples.
3. Process queries and return triples over HTTP in N3 format.
4. Maintain a thread for the waitToRead operation, which queries the space for a certain amount of time, if the triples searched for do not exist yet.
5. Generate and insert the metadata triples for timestamp and the author's IP, of written triples into a metadata subspace.

Due to the complexity of topics like security, transaction management and kernel interactions, these topics were not implemented for this space implementation. Because the aim of this thesis was to create a communication between a space an a mobile device, a special focus was laid on leaving as much calculation as possible to the space. This way the limited cpu speed of mobile devices should be saved.

As already mentioned the space was realized as a Java servlet [22]. Figure 3 shows the layer stack which composes the implementation.

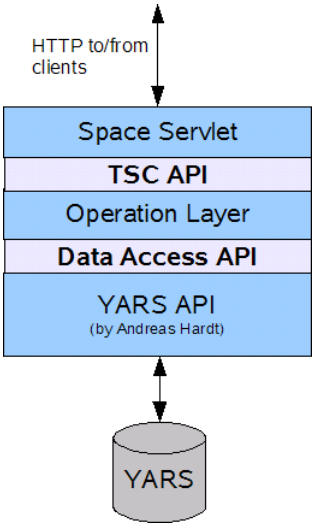


Figure 3: The implemented space layer stack

In a broad view it can be said that the servlet layer communicates with any possible clients over HTTP. The Operation Layer prepares the information to the right format so it can be stored in YARS. Finally the the Data Access Layer/YARS API stores or retrieves the data from YARS. The different layers are in charge of hiding their complexity from upper layers. For example when storing triples into the space, the Operation Layer simply uses operations from the Data Access API, knowing how the storing is done. The Data Access API simply defines two different operations, read and write. Those two operations are used by every TSC API operation to communicate with YARS. More on this layer can be found in Section 5.3.

The following part will describe the different layers, their functionality and their purpose in more detail.

5.1 The Space Servlet

Servlets react on HTTP GET, PUT, POST, DELETE and some other requests. In the case of this thesis, the space servlet takes GET requests for all commands that involve querying and returning triples, and POST requests to write triples into the space.

As seen in Section 4, request parameters are given to the servlet in the form of extensions to the servlet URI. When a new HTTP request is invoked on the servlet engine (see [22]), the server creates a new thread, which runs the servlet. According to the request which was made, the request is either forwarded to the *doGet()* or *doPost()* method inside the servlet code. This is

automatically done by the servlet engine. Inside these methods, the servlet checks which operation was requested through checking the *&method=* extension of the URI which invoked the servlet. According to the method the servlet then checks for more parameters which are needed to fulfill the operation. For example the read operation needs either a second parameter *&query=n3qlQueryString* or *&uri=URItoNamedGraph* to read triples from the space. If the appropriate parameters were not found an error message is sent back to the invoking client, with the response code 500 for "Bad Request". In the same way the parameter *&method=write* is demanded by the *doPost()* method of the servlet. When all required parameters were found, the servlet forwards the request to the appropriate operation layer method. In the case of the write operation the space servlet also creates a timestamp and retrieves the IP of the author of the triples. This information is needed to create the unique ID for the written triples, which will be further discussed below. The write operation in the Operation Layer then receives these parameters.

If no method was specified, the servlet assumes that the requester wants to retrieve triples from a URI which was appended to the servlet URI. For example one points his browser to *http://www.example.org/space/tripleID*. After receiving this GET request, the servlet retrieves the path extension *tripleID* and checks YARS for triples with this URI. If no triples were found, the servlet returns a "Not Found" message with the appropriate response code. Or if no ID was appended to the URI, a response message is sent indicating that the space is running. This way the space complies to the REST principle [6], which demands that every resource (in this case triples) has a unique URI which can be used to retrieve the resource.

After the Operation Layer has returned or caused an exception, the space servlet prepares a HTTP response message to the client. The response specifies the length of the data string, the data type, a response code and the data that is returned itself. When N3 triples are returned the data type is specified as *text/rdf+n3*. If a URI is returned the data type is specified as *text/uri-list*. The HTTP response code corresponds to successfulness of the requested operation. For example if a query did not match any triples, the response code is set to 404 "Not Found". If triples were found it is set to 200 "OK". Also if the request was malformed a 400 "Bad Request", and if there occurred an error inside the space or YARS a 500 "Internal Server Error" is returned.

5.2 Operation Layer

The operation layer is responsible for preparing the data which has to be inserted into the space. It is the space intern implementation of the TSC API (see Section 2.3). Depending on which operation is requested, different actions are made by the operation layer. These are explained below.

- If the requested operation is a **write** functionality, the operation layer takes the IP of the author of the triples, and the timestamp generated by the space servlet, to create the unique ID for the named graph to be written. How this ID is created will be explained further below. The triples are then passed to the YARS API, which uses the created ID as the location where to store the triples in the space. Also two metadata triples are created. One connecting the created ID with the timestamp and one connecting the ID to the author's IP address. These two triples are then passed to the YARS API and written into the subspace *http://www.example.org/space/metadata*. This allows, to add basic trust to triples as explained in Section 2.2. This topic is discussed in more detail in [20]. The created ID is then passed back to the space servlet.

- If the requested operation was a **read** functionality, requesting all triples from a given URI, the URI is passed to the to the YARS API and a *null* value is passed as the query string. This indicates to the YARS API that all triples from the given URI are supposed to be returned. The triples that YARS returned are passed back to the space servlet. If no triples were found, the *null* value is passed back.

- If the requested operation was the **read** with a query functionality, the query string is simply passed to the YARS API. The returned triples are then passed back to the space servlet. If no triples were found, the *null* value is passed back.

- If the requested operation is the **count** functionality, the query string is passed to the YARS API and the returned triples are counted. The number is then passed back to the space servlet.

- If the request operation was the **waitToRead** functionality, YARS is first queried with the specified query. If no triples were found, the thread sleeps for five seconds, and then queries the space again. This happens until the specified timeout is exceeded or the requested triples were found in that time. If matching triples were found , they are

normally passed back to the space servlet. If no triples were found, the *null* value is passed back.

As it is explained in the YARS Section 5.3, the YARS API throws N3Exceptions. These exceptions are not caught by the operation layer. Instead they are passed up to the invoking space servlet layer. They space servlet can then analyze the exception and then extract the response code from YARS, to see what went wrong. According to this, the space servlet can generate it's response to the client.

Unique ID creation

As already mentioned graphs of triples need a unique ID to make them globally identifiable and addressable as resources. The space implementation creates this ID with the IP of the writing client and a timestamp. First a hash code of the IP string of the author is generated. Then a slash is appended to the hashcode, and after this the timestamp. Author and timestamp together create a unique identifier for a certain write operation of a client. As seen in Section 3.2, this ID string is then appended to the URI of the YARS servlet. The returned URI could then look like this:

```
http://www.example.org/space/9342891244/213423423
```

Where the first number is the hash code over the IP of the author and the second the timestamp of the writing. The triples can then, be later retrieved through making a HTTP GET request on the space URI to which the ID string was appended. Because the hash code over the IP of the author is consistent, this number spans a subspace containing all triples written by a certain author. This would enable a client to retrieve everything written by a certain node. But at the time of this writing YARS unfortunately did not support the querying of subspaces.

5.3 Data Access API/Layer

As mentioned in Section 2.3, the Data Access API defines operations to store and retrieve data from a data store. It is supposed to abstract from any underlying lower level technology, which is needed to store and retrieve data from the data store YARS. In the case of YARS this lower level technology is HTTP. For this thesis, the Data Access API simply defines a read and a write operation. It was implemented using the YARS API already mentioned in Section 3.2. A short explanation of the usage of this API can be found in Appendix C.

5.4 A Space Interaction Example

In the previous sections the reference implementation TSC API and the implemented space were explained. Also the used storage system YARS was described in Section 3.2. This part finally brings these pieces together and shows how they interact with each other. Figure 4 shows this interaction. The numbers on the sides of the Figure correspond to the actions of the different components in time. The text below explains these actions, corresponding to the numbers. For this example the write operation was used.

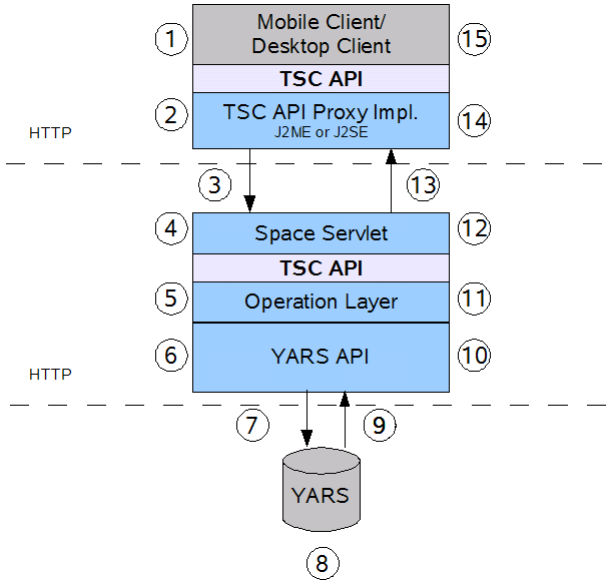


Figure 4: Client - Space - Storage communication

1. First a client invokes the

   ```
   URI graphURI = write(URI spaceURI,Vector triples);
   ```

   function from an application. Depending on the nature of this application, either the mobile or the desktop version of the API is used.

2. The implementation then prepares the space URI to connect to, adding the method parameter, adding the triples parameter and and encoding the URI to comply with the URI specification [8]. Since the method receives the triples as a Vector filled with triple objects, the method calls the *toN3()* method of all Triple objects, to construct the N3 string to send.

3. Then the connection to the space is established and the triples are transmitted over HTTP using a HTTP POST request.

4. Next the servlet checks for the method parameter. It finds that the write operation was requested, generates a timestamp, retrieves the IP of the client, retrieves the triple string and forwards the information to the operation layer.

5. The operation layer creates the unique ID from the timestamp and the authors IP address. To store the triples in the space the Operation Layer makes use of the YARS API of the underlying layer and specifies the created ID as the location where to store the triples. This layer also creates two metadata triples, connecting the unique ID with the timestamp and the author. It then calls the YARS API, which handles the storing of those triples.

6. The YARS API then performs a PUT request on the YARS servlet. After this the triples are transformed into bytes and sent to the YARS servlet via an output stream. The same procedure is done to store the timestamp and the author of the triple in the metadata subspace.

7. When the stream is opened the triples are transmitted over the network.

8. The YARS servlet gets the PUT request and opens an input stream reader. The triples are read and stored in the place according to the named graph ID appended to the YARS URI.

9. On success YARS returns a HTTP response with the response code 201 for "Created". This indicates that the resource was successfully added to the space.

10. The YARS API checks if no errors occurred and if the response code 201 was received. If this was so, it returns without throwing an exception.

11. Now the Operation Layer knows that no error occurred and the triples were inserted into YARS. It then returns the created ID to the URI of the space servlet and returns this URI.

12. When the space servlet receives the created URI it constructs the HTTP response to the client, also using the response code 201. Then an output stream is created to return the URI of the stored triples.

13. The response is sent via HTTP to the client.

14. On receiving the response from the servlet, the API implementation checks the response code. It finds out that the triples were successfully stored. Now it starts to read the input stream. It then creates an URI object from the returned URI string and passes it back to the invoking process.

15. The application now receives a URI where it can address the written triples as globally identifiable resources.

This example has shown the interaction between an arbitrary client, the space implementation and the storage system YARS. During all stages of this communication, errors can occur. But these errors would be caught and through response codes, and response messages indicated to the invoking client.

In this section we have seen how a complete client space interaction works in theory. It finalizes the description of the implemented space. After describing all parts of the space and API implementation, it is now possible to compare this setup to the proposed architecture of the TSC research group. This will be done by the following section. It will also analyze the overall output of this thesis.

6 Analysis

This thesis has shown how to create an interaction environment between a mobile device and a Triple Space implementation. But it has also shown that one carefully needs to choose common interfaces between J2SE and J2ME applications. Adopting to the requirements and limitations of mobile applications was the biggest challenge of this thesis.

In the following part the reference implementation and the proposed Triple Space architecture will be compared.

6.1 The Proposed Architecture Compared to the Reference Implementation

In this section the architecture which is proposed by the TSC research community will be compared to the implemented reference architecture. First the layer stack of the proposed architecture, which is outlined in [4], will be explained. Figure 5 shows this stack.

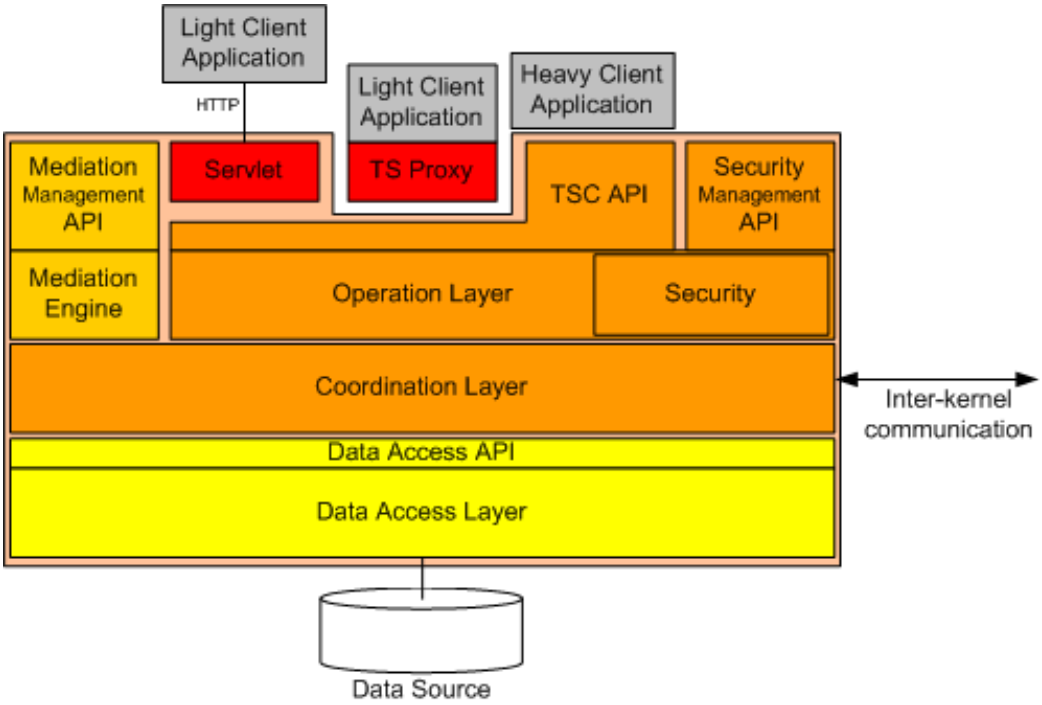


Figure 5: TTSC layer stack proposed in [4]

The possible forms of clients which can be seen in this figure have already been explained in Section 2.3. The different layers of this architecture will be explained in the following.

Security Management API

The Security API defines all security functionalities a space implementation needs to provide. The security issues regarding triple space computing are discussed in [20].

Operation Layer

This layer has the same responsibilities as the operation layer of the reference implementation of Section 5.2.

Coordination Layer

The coordination layer is responsible for transaction management. It also manages the synchronisation tasks with other connected spaces, which are

needed to ensure data homogeneity among all involved spaces. Replication and caching also other tasks which are managed by the Coordination Layer.

Data Access API

The Data Access API contains the common interface of functionalities which a space needs to implement to access a data source. It only contains simple read, write and delete functionality.

Data Access Layer

Implements the functionalities described in the Data Access API. The implementation details are different for all underlying storage facilities. For example different database implementations or even a simple file could be used. The implementation details for this layer in the reference space implementation can be found in Section 5.3.

Data Source

Is a specific data storage and retrieval facility. Could be any database implementation or even a simple file. In the case of the reference implementation for this thesis, the RDF store YARS was used. More information on this technology can be found in the background technology Section 3.

Mediation Management API

The Mediation Management API defines functionalities to provide homogeneity in the data formats involved in the space. These functionalities are responsible for mapping between data representation formats. This should ensure interoperability between involved peers. In particular it should also take care of adapting the data to the preferences, and and dynamic conditions of a device. This becomes especially of importance in the context of ubiquitous computing, where many different devices with different capabilities interact with each other.

Mediation Engine

The Mediation Engine implements the functionalities described in the Mediation Management API.

Comparing Figure 2, with Figure 5 of the proposed architecture, it can be seen that the proposed architecture foresees three different forms of clients.

A light client, communicating to the space via HTTP, another light client as a proxy implementation and a heavy client that can also take part in storing data. In contrast, the only proxy implementation was chosen for the reference implementation.

When comparing the proposed TSC API and the implemented TSC API, in sections 2.3 and 4, also many differences can be seen. First the naming of the operations. For example in the implemented TSC API the *read* operation is overloaded to either query the space or read triples from a certain URI. This is due to the fact that the research team renamed the operation, when the reference implementation was already implemented. Some operations were omitted to implement. Others, like *subscribe* or *advertise* were far out of the scope of this thesis.

Figure 3 shows the implemented layers forming the space. Here can be seen that the layers concerning mediation and security were left out in the reference architecture. Also the coordination layer, which coordinates the communication between connected spaces, was not implemented. Hence the reference architecture does not follow the super-peer model mentioned in Section 2.3. Also transaction management was not implemented. As far as creating a unique ID for inserted triples and supporting blocking operation like *waitToRead* the two operation layers can be compared. But as mentioned above, higher level functionalities like *advertise* or *subscribe* were not implemented.

7 Conclusions

This thesis gave a broad overview of the current state of research in Triple Space Computing. After explaining the background technologies, the implementation details of two TSC API reference implementations were explained in Section 4. Moreover, the architecture of the implemented Triple Space was described. It consists of a servlet, which communicates with clients over HTTP. This minimal architecture enables clients to store and retrieve RDF triples from the space, and associates every written graph with a URI. This makes it possible to address graphs as resources in a network. As in the previous section, the implemented setup is far away from a complete clustered space implementation. But some functionalities would be easy to implemented, when YARS becomes more advanced. For example, if YARS would support relative path querying, it would be possible to implement space mapping inside of YARS. This means that the path extensions of triples URIs would enable YARS to host many spaces.

Having explained all the technologies involved, a complete interaction

example between a client and the space was given. The analysis finally showed that a minimal space architecture was achieved, and stated which parts are missing, to fully comply with the current state of research in TSC.

Acknowledgments

The author of this thesis would like to thank Mark Mattern for being a reliable and proficient partner, and also Reto Krummenacher for supervising this thesis.

References

[1] Tim Berners-Lee. *Notation 3 - An readable language for data on the Web.* `http://www.w3.org/DesignIssues/Notation3.html`, 2006.

[2] W3 Bert Bos. *URI encoding programs.* `http://www.w3.org/International/0-URL-code.html,License:http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231`, 2002.

[3] Ch. Bussler. *A Minimal Triple Space Computing Architecture.* 2nd WSMO Implementation Workshop, Innsburck, Austria, 2005.

[4] Chris Bussler, Edward Kilgarriff, Reto Krummenacher, Francisco Martin-Recuerda, Ioan Toma, and Brahmananda Sapkota. *D21.v0.1 WSMX Triple-Space Computing.* `http://www.wsmo.org/TR/d21/v0.1/`.

[5] W3C World Wide Web Consortium. *Extensible Markup Language (XML).* `http://www.w3.org/XML/`.

[6] R.T. Fielding and R. N. Taylor. *Archtictural systels and the design of network-based software architectures.* University of California, Irvine, 2000.

[7] D. Gelernter. *Generative Communication in Linda, ACM TOPLAS, 7(1).* 1985.

[8] Network Working Group. *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax.* January 2005.

[9] Andreas Hardt. *Introduction to the YARS API.* `http://sw.deri.org/2004/06/yars/trunk/doc/java-api.html`, 2006.

[10] Andreas Harth, Matteo Magni, and Stefan Decker. *Scalable Distributed RDF Storage Infrastructure*. June 2005.

[11] J.J.Carroll, Ch. Bizer, P. Hayes, and P Stickler. *Named Graphs*. Journal of Semantics 3(4), 2005.

[12] G. Klyne and J. J. Carroll (eds.). *Resource Description Framework: Concepts and Abstract Syntax. W3C Recommendation, February 2004*. `http://www.w3.org/TR/rdf-concepts/`.

[13] Reto Krummenacher, Ying Ding, Edward Kilgarriff, Brahmananda Sapkota, and Omair Shafiq. *Specification of Mediation, Discovery and Data Models for Triple Space Computing, TSC project deliverable*.

[14] Reto Krummenacher, Martin Hepp, Axel Polleres, Christoph Bussler, and Dieter Fensel. *WWW or What Is Wrong with Web Services*.

[15] Reto Krummenacher, Jacek Kopecky, and Thomas Strang. *Sharing Context Information in Semantic Spaces, Proc. of the OTM 2005 Workshop on Context-Aware Mobile Systems (CAMS'05):Agia Napa, Cyprus*. October 2005.

[16] Reto Krummenacher, Francisco J. Martin-Recuerda, Martin Murth, and Johannes Riemer. *D1.2 v1.0 TSC Framework*. `http://tsc.deri.at/internals/D12.html`.

[17] Reto Krummenacher, Thomas Strang, and Dieter Fensel. *Triple Spaces for a Ubiquituos Web of Services*. Position Paper: W3C Workshop on the Ubiquitous Web, 2005.

[18] Mark Mattern. *TSC Implementation for Mobile Devices*. Bachelor Thesis, University Innsbruck, 2006,.

[19] Sun Microsystems. *Java Technology: Java 2 Platform, Micro Edition (J2ME)*. `http://java.sun.com/javame/index.jsp`.

[20] M. Murth and J. Riemer. *Security and Privacy Models in Triple Space*. TSC Project Deliverable D1.4 v1.0, 2006.

[21] Oracle. *Berkeley DB Java Edition*. `http://www.sleepycat.com/products/bdbje.html`.

[22] SUN. *Java Servlet Technology*. `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html`.

[23] SUN. *SUN - Java Community Process*. http://www.jcp.org.

[24] Tomcat. *Apache Tomcat servlet container*. http://tomcat.apache.org/.

[25] W3.org. *N3QL - RDF Data Querying Language*. http://www.w3.org/DesignIssues/N3QL.html.

[26] B. Yang and H. Garcia-Molina. *Designing a Super-Peer Network*. 2003.

A Using a Worker Thread in J2ME or J2SE

An application using one of the mentioned TSC API implementations should use a worker thread. This leaves the main thread idle for user interaction. How such a thread is invoked can be seen below. This code was used in the mobile and desktop reference clients made for the Bachelor Thesis "TSC Implementation for Mobile Devices" by Mark Mattern [18].

```
1  /* Application trying to retrieve some triples
2  * from the space.
3   */
4  ...
5  new Thread() {
6          public Vector returnVector = null;
7          //define n3 query string
8          public String queryString = ...
9          public void run () {
10                 try {
11                 returnVector = spaceAcc.read(spaceURI,queryString);
12                 ...
13                 /* Do something with the returned data,
14                 * for example update GUI.
15                 */
16                 } catch (IOException e) {
17                         ...
18                         }
19                 }
20         }.start();
```

B Triple UML

To conveniently work with RDF data in a Java environment, it is necessary to implement classes which represent the data, instead of working with RDF

strings. The UML diagram below shows how the *TripleImpl* class was implemented for J2ME version of the triple implementation. This architecture is analog to the implementation in the YARS API, but does use technologies which are not available in J2ME. These are for example the class *ArrayList* or the interface *serializable*.

As it can be seen in the UML diagram, an RDF triple consists of three Nodes. One being the subject, one being the predicate, one being the object. Resources and literals need to implement the methods required by the *Node* interface. The most important method of the *Node* interface is the *toN3()* method. It converts a node it to its N3 representing string. While subject and predicate have to be resources, the object can either be a resource or a literal. As it can be seen in the diagram, the literal implementation also allows for typed literals. This means that to the data of the literal, also language and data type information can be attached. Both are also identified by URIs. The *TripleImpl* class extends the *Resource* class. This means that a triple is also a resource, and can therefore be identified by an URI.

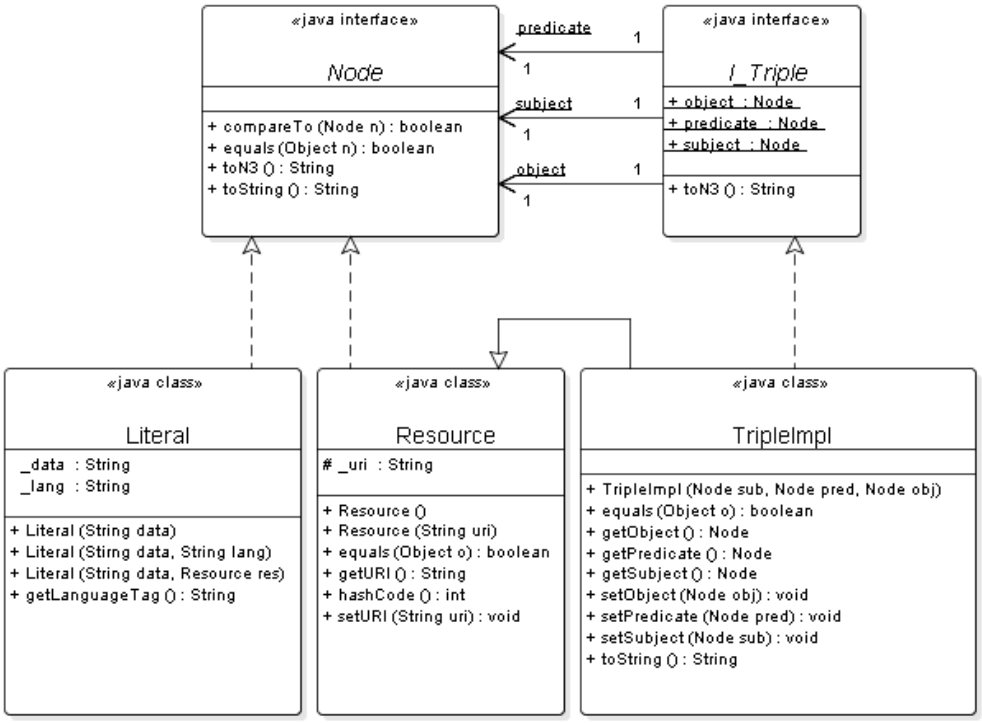


Figure 6: UML diagram of the triple implementation

C YARS API

First the client needs to establish a connection to YARS with:

```
Connection con = DriverManager.getConnection(String yarsURIString);
```

After that, a statement has to be created. This statements needs the context in which the statement should be executed as an argument. The context is also referred to as the ID of the written triples, and can be seen as the path to the triple inside YARS.

```
Statement stmt = con.createStatement("context");
```

When the statement is created, different actions like query, write or delete can be performed on this context. This is done as follows.

```
stmt.executeInsert(String tripleString);
stmt.executeDelete(String n3qlQueryString);
ResultSet rs = stmt.executeQuery(String n3qlQueryString);
```

With these three methods all possible actions can be performed on YARS. The method *executeQuery* returns a *ResultSet* object containing the found triples.

All actions can throw N3Exceptions which extend the class Exception. Through invoking the method *getResponseCode()* of a thrown N3Exception, it can be found out what went wrong. More detailed information about the usage of the API can be found here [9].