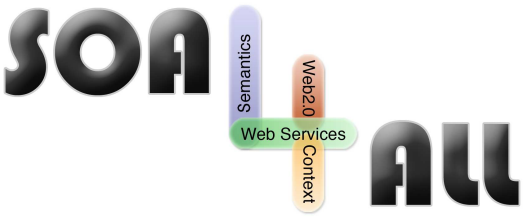

Project Number:	**215219**
Project Acronym:	**SOA4All**
Project Title:	**Service Oriented Architectures for All**
Instrument:	**Integrated Project**
Thematic Priority:	**Information and Communication Technologies**

D1.3.2B Distributed Semantic Spaces: A First Implementation

Activity N:	Activity 1 – Fundamental and Integration Activities	
Work Package:	WP1 – SOA4All Runtime	
Due Date:		M18
Submission Date:		14/09/2009
Start Date of Project:		01/03/2008
Duration of Project:		36 Months
Organisation Responsible of Deliverable:		UIBK
Revision:		1.1
Authors:	Reto Krummenacher UIBK Michael Fried UIBK Fabrice Huet INRIA Imen Filali INRIA Laurent Pellegrino INRIA Christophe Hamerling eBM	
Reviewers:	Dong Liu OU Yosu Gorronogoitia ATOS	

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	**X**

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	2009-07-15	First TOC	Reto Krummenacher (UIBK)
0.2	2009-08-03	First draft	All
0.3	2009-08-10	Updated draft	Reto Krummenacher (UIBK), Fabrice Huet (INRIA), Michael Fried (UIBK)
0.4	2009-08-17	Web service bindings	Christophe Hamerling (EBM)
0.5	2009-08-18	Pre-final version	Reto Krummenacher (UIBK)
0.6	2009-08-19	Internal release	All
1.0	2009-09-08	Final release for submission	All
1.1	2009-09-14	Final editing	Malena Donato (ATOS)

Table of Contents

EXECUTIVE SUMMARY 5

1. INTRODUCTION 6

1.1 PURPOSE AND SCOPE 6
1.2 STRUCTURE OF THE DOCUMENT 6

2. REFLECTION ON THE SPECIFICATION 7

2.1 REDUNDANCY IN SEMANTIC SPACES 7
2.2 UPDATE FUNCTIONALITY 8

3. SOFTWARE DESCRIPTION 10

3.1 SEMANTIC SPACES API AND SERVICE BUS INTEGRATION 10
3.1.1 Semantic Space Core 10
3.1.2 Space WS Binding 13
3.2 PEER-TO-PEER DISTRIBUTION AND INDEXING INFRASTRUCTURE 15
3.2.1 A generic overlay library 16
3.2.2 CAN implementation 16
3.2.3 Chord implementation 18
3.3 PERSISTENCY LAYER AND OWLIM BINDINGS 20
3.3.1 tsontology.owl 22

4. INSTALLATION AND CONFIGURATION 24

5. PERFORMANCE AND EFFICIENCY CONSIDERATIONS 26

5.1 TRACKER SCALABILITY AND RESILIENCE 26
5.2 OPTIMIZED MERGE OPERATION 26
5.3 IMBALANCE IN DATA STORAGE 27
5.4 ASYNCHRONOUS PROCESSING OF QUERIES 28

6. CONCLUSIONS 29

7. REFERENCES 30

List of Figures

Figure 1: Architecture of a semantic space....10
Figure 2: Semantic Space Core....12
Figure 3: Space implementation as Web service....14
Figure 4: Space to Web service binding....15
Figure 5: Space to Overlay layer....16
Figure 6: Example of join....17
Figure 7: Successive splitting....17
Figure 8: Example of conjunctive query....18
Figure 9: Example of a Chord Overlay....19
Figure 10 : Finger tables....20
Figure 11 : Relation between kernel, space and data store....21
Figure 12: Query and Response interfaces....21
Figure 13 : Internal classes for messages in the CAN overlay....22
Figure 14 : Example of merging without optimization....26
Figure 15: Example of optimized merging....27
Figure 16: Imbalance in data storage....27

Executive summary

This deliverable describes the first implementation of the distributed semantic spaces infrastructure. The implementation yields the realization of the concepts and specifications that were released with deliverable D1.3.2A Distributed Semantic Spaces: A Scalable Approach To Coordination [1]. As such, this deliverable will also shortly reflect on the content of this past report in order to clarify some of the questioned points.

1. Introduction

This report on distributed semantic spaces describes the first implementation of the semantic spaces prototype. In more detail, the implementation presented in this deliverable yields the realization of the concepts and specifications that were released with deliverable D1.3.2A Distributed Semantic Spaces: A Scalable Approach To Coordination [1]. The prototype delivers the space logic implemented and deployable on top of the distributed ProActive Grid infrastructure, and provides the space-specific parts of the SOA4All Distributed Service Bus. This implementation thus complements the DSB prototype that is shipped with deliverable D1.4.1B SOA4All Runtime [3]. Therefore, this deliverable focuses on the semantic spaces infrastructure only. Further details about the incorporation with the bus and the integrated deployment, configuration and use are given in the main architecture implementation document (cf. deliverable D1.4.1B).

1.1 Purpose and Scope

The goal of this deliverable is to provide a written description of the distributed semantic spaces prototype (D1.3.2B Distributed Semantic Spaces: A First Implementation). We refer the reader to D1.3.2A for details about technical background and baseline, as well as the architecture and design of the prototype implementation.

In addition to the information about the implementation, this deliverable yields a reflection of some of the decisions and results that were presented in D1.3.2A, and as such complements the updated release of the conceptualization and specification report of month M12. For this reason, this deliverable also contains a section on performance and efficiency considerations, as it was desired by the reviewers (Section 5).

1.2 Structure of the document

In order to best respond to the purpose of the deliverable, we structure the document into the following sections. Firstly, in Section 2, we re-consider some of the decisions that were taken in the specification of the semantic space infrastructure. This concerns mainly the issue of redundancy across subspaces, and the question of a potential update operation. In this section, we discuss our decisions and reflect on the choices made. The software release itself is presented in Section 3. The section is divided into three parts that consider a) the interaction model and API implementation, b) the indexing and distribution mechanisms based on P2P technology, c) the realization of the persistency layer by means of OWLIM. In Section 4 we present installation and configuration details. This part of the deliverable complements the corresponding section on installation and configuration in deliverable D1.4.1B [3]. Before concluding the deliverable with Section 6, we dedicate Section 5 to the discussion of performance and efficiency-related considerations in terms of the P2P-based implementation of the SOA4All semantic space infrastructure.

2. Reflection on the Specification

The specification of the SOA4All semantic space infrastructure has raised a few technical concerns. Rather then providing an update of the specification deliverable, we will explicitly address the main two issues in this section. The first issue that we will discuss concerns redundancy of data within multiple subspace hierarchies, as they were specified for the semantic spaces in Deliverable D1.3.2A [1]. The second reaction was triggered by the missing update operation that would be expected from a database-like infrastructure in order to match the generally available CRUD (create, read, update and delete) functions.

2.1 Redundancy in Semantic Spaces

Redundancy in the context of semantic spaces refers to the maintenance of copies of syntactically or semantically identical triples that were published by users to the space infrastructure. Although the semantic space infrastructure allows for publishing identical RDF triples to different subspaces, this does not create any redundancies. Such triples are not perceived as copies but as individual pieces of information. The situation is different if the triples are published to the same space. In those cases, the triples are considered the same, and only one copy is maintained. This is supported by the insight that doubling a truth-value does not make the statement any truer in the given context.

More technically spoken, subspaces create isolated containers for the sharing of semantic artifacts – the implications of sub-spacing is discussed later in this section. Data that is published to different spaces is thus always virtually isolated from the data in other spaces, and in consequence perceived as a different piece of information.[1] Semantic relationships are only exploited within a given space, and not across the boundaries of virtual subspaces. In that sense, also identical, the RDF triples are not regarded as copies, and do not cause any redundancy problems.

While there is thus no detectable problem with redundancy in the context of disjoint virtual spaces, the situation is slightly more complex when looking at space hierarchies and federations. According to the specification, any two spaces can be brought in a part-of relationship that makes sub-spaces become parts of the super-space in terms of the published data too. All queries expressed to a super-space are in turn resolved against the data of the entire sub-tree unless a client explicitly states that non-recursive retrieval is required (cf. Section 2.2.2 in D1.3.2A).

A priori the same regulations apply, as they were discussed above. This is a direct consequence of the fact that publication is always targeted to a particular space instance. Subspaces have no influence in terms of publishing and maintenance of triples, and data that is published to different spaces is always stored as different pieces of data together with the space identifier. Sub-spaces and federations are only influential at retrieval time, and it is up to the space implementation to take care of redundant triples that might match a given query. This is however clearly a per query process and does not influence the data that is shared in other spaces, as retrieval does not alter the state, nor remove a triple. The same accounts for federations that are temporary read-only containers, thus similar to spaces in terms of read access to data.

The last aspect that has to be considered in the context of redundancy is related to operations that alter the state of a space in regards to the contained data. According to Table 2 in D1.3.2B, the semantic spaces interaction model knows one operation for removing data from a space:

[1] Note that the write operation requires a space identifier to target a given piece of data to a particular subspace in which the data is maintained and processed.

remove(URI space, Query query): Set<Triple>	Same as the basic query operation, however this operation does not only return the matching triples, but removes them from the given space.

As triples that are stored in different spaces are treated as different pieces of data, there is no cross-space consequence that has to be taken into account when a remove operation is invoked. In this respect, it is important to note, that similar to the write operation, removal requires the indication of a target space from which the matching data shall be deleted. This ensures that only the triples in the given space are affected, and hence no synchronization with data maintained in other spaces is necessary.

In summary, the specified space and interaction models do not imply any redundancy problems, as the existence of multiple versions of the syntactically same triple only affects the execution of retrieval operations. Neither at publication nor removal time, any other spaces than the targeted one, and thus no other data, is affected by the execution of the operation.

2.2 Update Functionality

The interaction model of the SOA4All semantic space infrastructure, as defined in deliverable D1.3.2A, knows operations for publishing semantic data in form of RDF triples, and for various ways of retrieving parts of the shared data. In addition, a simplified removal operation operates over explicitly published data in a particular space. In order to establish the whole set of CRUD operations, semantic spaces needed to install an update operation too. There are several reasons for why this was not done when the infrastructure was specified in the first year of SOA4All.

One of the prime reasons is simplicity. The goal of the interaction model is to provide a small and simple set of operations to interact over shared semantic data. The update functionality is in principle a combination of a remove call and a re-publish operation that should be transactionally bound, in order to ensure atomicity across an entire semantic space. This view on the update operation indicates two facts for which the update operations was neglected: i) update can be primitively realized by means of the remove(URI space, Query query) and write(URI space, Set<Triple>) methods, and ii) a clean implementation would require support for transactionality which currently is not provided nor planned. In fact, transactionality, although it significantly increases the expressiveness of the interaction model, does clearly hamper the scalability expectations of the semantic space infrastructure in distributed scenarios. As none of the currently known clients to the semantic spaces requires transactions (SOA4All Platform Services or the use case applications), it was preferred to concentrate on the core publish and read operations that have the most promising consequences in terms of scalability and performance.

Re-considering the user requirements that were expressed by the developers of the platform services, and the providers of the use case demonstrators reveals that updatability is not (yet) a required feature. Generalizing this absence of need in the context of the current status of SOA4All, resulted in not having an update operation specified for the semantic spaces infrastructure.

In the following, we shortly discuss the most prominent usage scenarios of semantic spaces in the SOA4All that are the storing of service descriptions, and the sharing of monitoring data. Further uses such as the sharing of user profile data might have new requirements and will have to be considered during the refinement phase of the semantic space infrastructure specification that is due between the months M13 and M18 of the project.

Storing semantic service descriptions: The storage of semantic service descriptions, and similarly the management of semantic process descriptions, is governed by the service registry component, respectively a process registry. In terms of updatability, it is important to

note that differing descriptions about the same service implementation are treated in SOA4All (per definition) as different Semantic Web services. In other words, no matter if the differences result from a different view point by another annotator, or because of an updated version of a description, the former release of the Semantic Web service will not be altered (though possibility deleted) but kept in parallel in the registry. In consequence, the management of semantic service description and processes in SOA4All does not require any update functionality.

Sharing monitoring data: The monitoring platform of SOA4All is described in the deliverables D1.4.1A [2], respectively D1.4.1B [3] in what concerns the implementation. Monitoring is a continuous process and the different monitoring components store up to date information to the space infrastructure. An update of a monitoring state, is in fact a new snapshot of the current status of the observed entity, be it infrastructure, communication or service endpoint qualities. Although, we refer to updates of states here, in terms of the stored data, it is not an update in the CRUD sense, but the addition, and thus the publishing, of additional information about the observed object. Again, publishing and sharing monitoring data does not require an update operation, but only extends the monitoring data graph.

In summary, it is correct that the operations which are defined for semantic spaces could include an update method in order to cover the whole range of CRUD functionalities. This is not yet the case, as the focus of the interaction model was to cover the requirements of the space users with an as simple and as small set of operations as possible. Still, we will have to reconsider the set of operations based on the results of the first integrated prototype of the SOA4All infrastructure. Insights gained from this first consolidated realization, and new requirements from various platform services, the SOA4All Studio or use case implementations, might in fact reveal the need for an update operation that goes beyond the currently simplified approach with remove and write. A follow-up investigation of this matter will be released with deliverable D1.3.3A in month M24.

3. Software Description

This section describes the semantic spaces software package. In order to provide a well-structured approach to the software, we divide the overall software package into three parts:

- API specification and mappings for the service bus integration,
- Peer-to-peer distribution and indexing infrastructure, and
- Storage layer bindings to BigOWLIM.

The corresponding implementation architecture is shown in Figure 1 below.

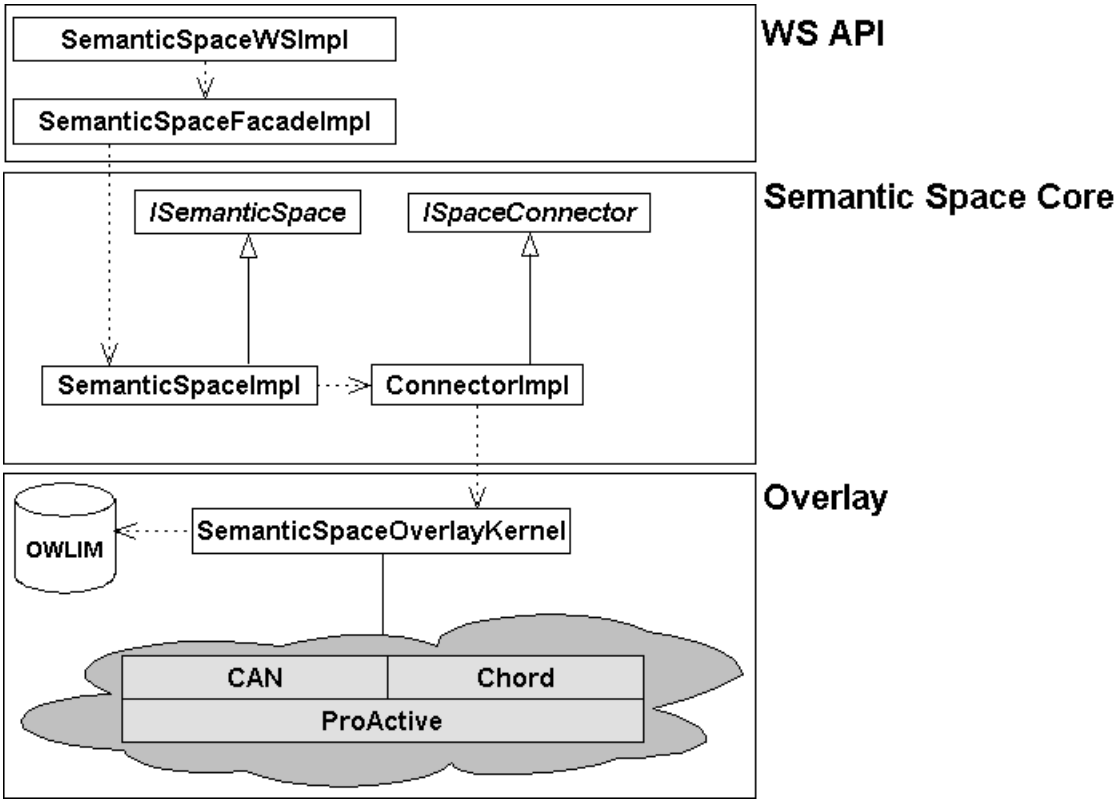

Figure 1: Architecture of a semantic space

The combination of these three components is referred to as a kernel from now on.

3.1 Semantic Spaces API and Service Bus Integration

The Semantic Spaces API offers multiple operations to perform various queries on the underlying persistency layer. The implementation follows the specification in Section 2.2 of Deliverable D1.3.2A. The core functionality in terms of implementation will be explained in this section.

3.1.1 Semantic Space Core

The Semantic Space API is realized by the ISemanticSpace interface (cf. Figure 2), which groups the various operations of the semantic space into three functional areas:

1. Space management operations:

   Space management operations handle the creation and deletion of spaces as well as the relations between spaces. The following methods described in deliverable

D1.3.2A have been implemented:

- createSpace(URI space): creates a new space
- createSpace(URI space, URI parent): creates a new space which is a child of parent
- joinSpace(URI space): joins a space
- leaveSpace(URI space): leaves a previously joined space
- listJoinedSpace(): lists all currently joined spaces
- URI createFederation(Set<URI> spaces): combines multiple spaces to a single space URI to query them at once
- deleteFederation(URI fed): deletes a previously created federation
- createSpaceRelation(URI sSpace, URI relation, URI oSpace): creates a relation between spaces. The relations that are part of the ontology of deliverable D1.3.2A can be expressed when specified as given in Section 3.3.1.

2. Storage access operations:

   Per specification, RDF statements can only be written to a previously joined space. The remove operation also returns the deleted triples. The following methods described in Deliverable D1.3.2A (Section 2.2 and Section 3.2) have been implemented for the current prototype release:

   - write(URI space, Set<Statement> triples): writes a set of triples to a previously joined space.
   - Set<Statement> remove(URI space, Query query, long maxTimeout): removes and returns the queried triples from only the given space.

3. Query operations:

   The space implementation accepts two different types of requests. First, the retrieval operations can be called by means of standard SPARQL SELECT or CONSTRUCT queries. These queries result either in sets of triples to be returned (construction) or in a set of bound variables according to the query, in the case of SELECTION. A second possibility that the space implementation offers, is the specification of the data to return by means of more simple triple patterns. The simplicity of the triple patterns not only eases the specification of a request, but also the resolution of the request within the distributed space infrastructure. Triple patterns do not require the manipulation of joins or optional patterns as they are offered by the SPARQL specification. While SPARQL queries are directly passed on to the underlying distributed storage platform, triple patterns are adapted by the internal Query Logic Component to match the SPARQL syntax before being forwarded. For instance, the triple pattern “?s ?p ?o” will be transformed to the following SPARQL query:

```
CONSTRUCT {?s ?p ?o}
WHERE {
        GRAPH <spaceURI>
        {?s ?p ?o} . }.
```

   As stated above, the query operations are not only divided in two subgroups in regards of the query specification formalism, but also in terms of the applied query type. The operation ‘query’ supports SPARQL CONSTRUCT and triple pattern queries, while the ‘queryV’ operations support SPARQL SELECT queries and

bind the variables (v) of the request in an answer set.

The following methods have been implemented:

- Set<Statement> query(URI space, Query query, long maxTimeout): queries a space and all its subspaces.
- Set<Statement> queryNonRecursive(URI space, Query query, long maxTimeout): queries only the given space.
- Set<Statement> query(Query query, long maxTimeout): queries all spaces.
- TupleQueryResult queryV(URI space, Query query, long maxTimeout): queries a space and all its subspaces.
- TupleQueryResult queryVNonRecursive(URI space, Query query, long maxTimeout): queries only the given space.
- TupleQueryResult queryV(Query query, long maxTimeout): queries all spaces.
- Set<Statement> query(URI space, URI relation, Query query, long maxTimeout): queries a space and all spaces in given relation to this space.
- TupleQueryResult queryV(URI space, URI relation, Query query, long maxTimeout): queries a space and all spaces in given relation to this space.

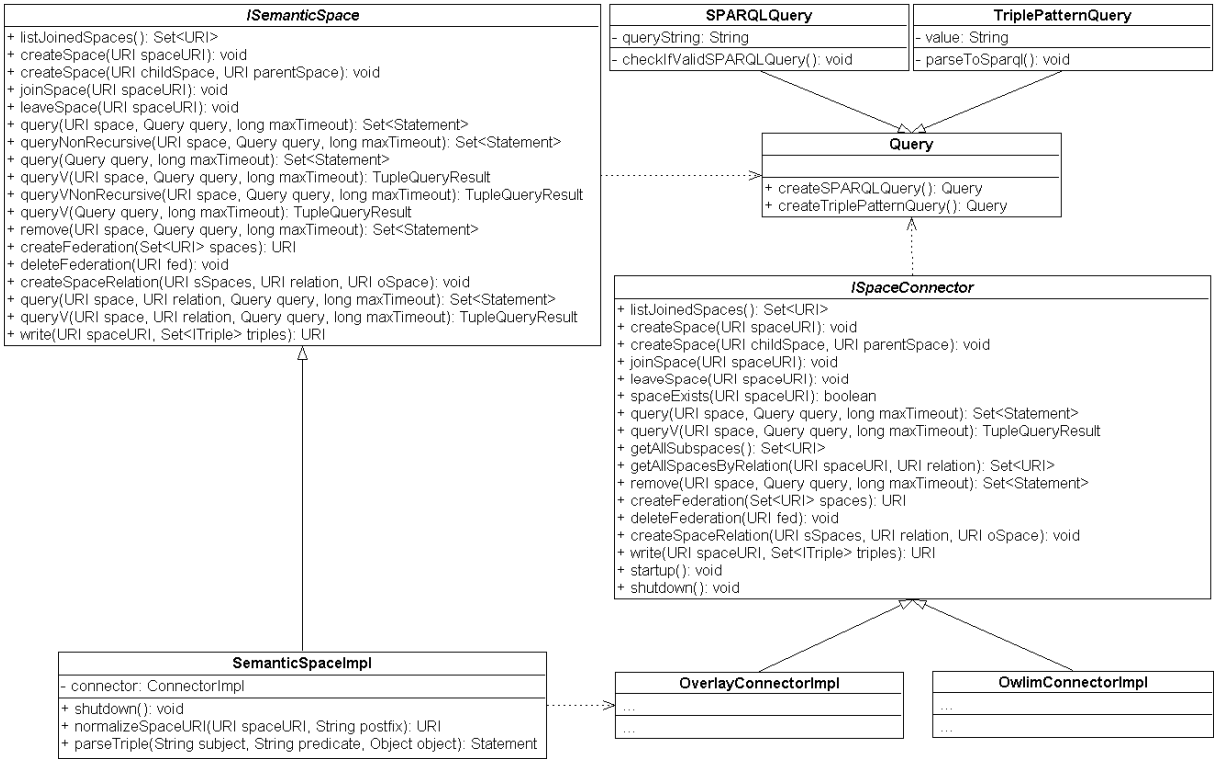

Figure 2: Semantic Space Core

All the query operations, as well as the remove operation are blocking; i.e., they only return if matching triples are found. Either the matches must already be published at request time, or newly written triples appear while the retrieval operation is still pending. In order to avoid deadlocks, the operations accept a timeout parameter, and return with an empty result set

when reaching the maximum timeout without discovering any data that matches the query. In other words, an empty result set suggests that the indicated timeout has been reached and that no matching data has been was discovered prior to the expiration of the timeout period. However, if the timeout was set to 0, a non-blocking request is performed, and the empty result set is suggesting the non-existence, or rather non-discoverability of matching data.

The query(URI federation, Query query) and queryV(URI federation, Query query) methods have not been implemented separately. As already stated in Deliverable D1.3.2A, they are redundant in terms of the data source identifier (the URI space and the URI federation are the same from a technical point of view). The federation specific operations are thus covered by the operations query(URI space, Query query) and queryV(URI space, Query query) respectively.

With respect to exception handling, the actual implementation extends the specification from Deliverable D1.3.2A. All exceptions are defined in the semantic space core implementation and are all derived from one generic top level semantic space exception. An exception is thrown when a SPARQL query is not valid or a triple pattern is not formatted according to the pattern “?s ?p ?o”. It is also prohibited to create an already existing space or to perform any operation on a non-existing space. Additionally an exception occurs when writing triples to a space that has not been joined to.

The following exceptions can occur when the Semantic Space API is accessed and these are, as mentioned above, all derived from the generic top level SemanticSpaceException:

- QueryStringException: to indicate that a SPARQL query is not valid or a triple pattern is not formatted according to the standard RDF pattern; e.g., ?s ?p ?o.
- SpaceException: to indicate a problem in the execution of the semantic space logic; this is a generic space exception.
- SpaceAlreadyExistsException: to indicate that the caller is trying to create a space that already exists.
- SpaceNotExistsException: to indicate that a caller is trying to perform any operation on a space that does not exist.
- SpaceNotJoinedException: to indicate that a caller is trying to write to a space that was not joined by the selected kernel; i.e., the kernel that is used to execute the publication of some semantic data.

3.1.2 Space WS Binding

At the level of the SOA4All Distributed Service Bus, the Semantic Space API is exposed as a standard Web service with the help of the DSB integration facilities. The currently deployed version that enables the implementation presented above is published at:

http://***<HOST>***/petals/services/SpaceService?wsdl,

where ***<HOST>*** refers to a SOA4All DSB node.

For example:

http://SOA4All.ebmwebsourcing.com/petals/services/SpaceService?wsdl
http://SOA4All-runtime.sti2.at/petals/services/SpaceService?wsdl

The space implementation is exposed to DSB clients as a Web service and uses the DSB integration facilities to do so; i.e., the space is treated the same way as SOA4All platform service.

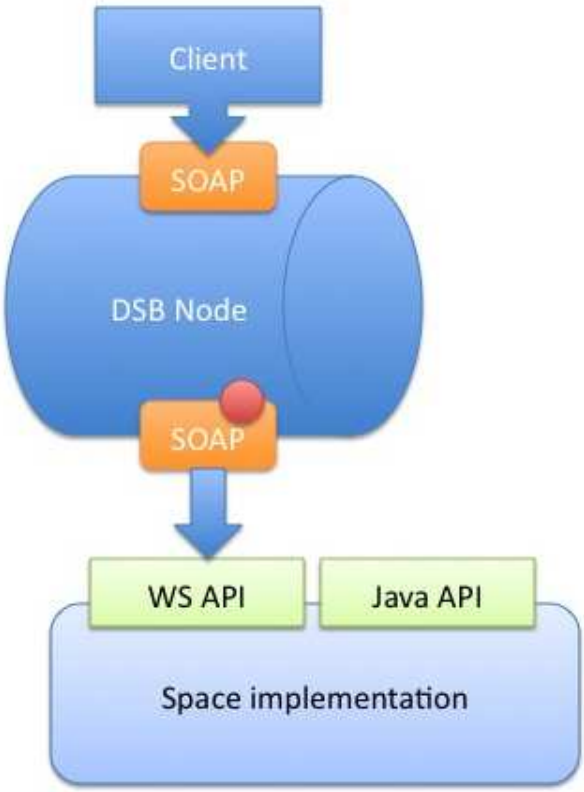

Figure 3: Space implementation as Web service

In Figure 3, it can be seen how the space implementation runs in a separate software module, as the DSB node. This implementation provides different types of API. The Web service API is bound to the service bus with the help of the SOAP binding component provided by the Distributed Service Bus. Once bound, the Web service exhibits the characteristics of a platform service at the level of the DSB. This means that all platform services can use it, and that it is exposed to bus-external application and business services via that binding component. In the current integration, the space service (bullet in Figure 3) is exposed as a Web service to external DSB clients. This provides the capability to:

- manage and monitor messages which are exchanged between consumers and providers
- update the space implementation without any impact at the consumer side (platform services or external clients)
- move the space implementation runtime location. If the space implementation is bound to another DSB node, it remains accessible at the same DSB endpoint address.

The Web service implementation is split into two parts, the API realization and the service functionality:

1. The WS-API consists of the following classes:
    - BindingTO is a helper class to install name to value bindings. This is needed to map the Web service protocol onto OWLIM classes, as the Web service cannot communicate those objects directly.

- BindingValuesResultTO is a helper class to wrap the variable bindings that are returned as result of a queryV operation
- HashMapAdapter for the marshall and unmarshall bindings.
- HashMapEntryType
- HashMapType
- SemanticSpaceException is the Web service-side equivalent to the exceptions of the semantic space implementation. This exception is thrown if an exception occurs during a space access via the Web service interface.
- SemanticSpaceWS is the JAX-WS interface of the Web service.
- TripleTO is used to convert RDF statements of the space implementation to a purely String-based representation. As above, this is necessary, as the Web service cannot communicate OWLIM objects directly.

2. The Web service implementation uses the Apache CXF framework (http://cxf.apache.org), which is an open source services framework. CXF helps to build and develop services using front-end programming APIs, like JAX-WS. These services can communicate via a variety of protocols such as SOAP, XML/HTTP, RESTful HTTP, or CORBA. The implementation consists of the following classes:
   - SemanticSpaceFacade is a mirror interface ISemanticSpace, but relies solely on standard Java classes that are compatible with the Web service communication protocols.
   - SemanticSpaceFacadeImpl handles the conversions and mappings between the methods specified in the SemanticSpaceFacade and the actual semantic space interface ISemanticSpace that uses OWLIM-specific classes.
   - SemanticSpaceWSImpl is the actual Web service implementation that is accessed via the SemanticSpaceFacade.

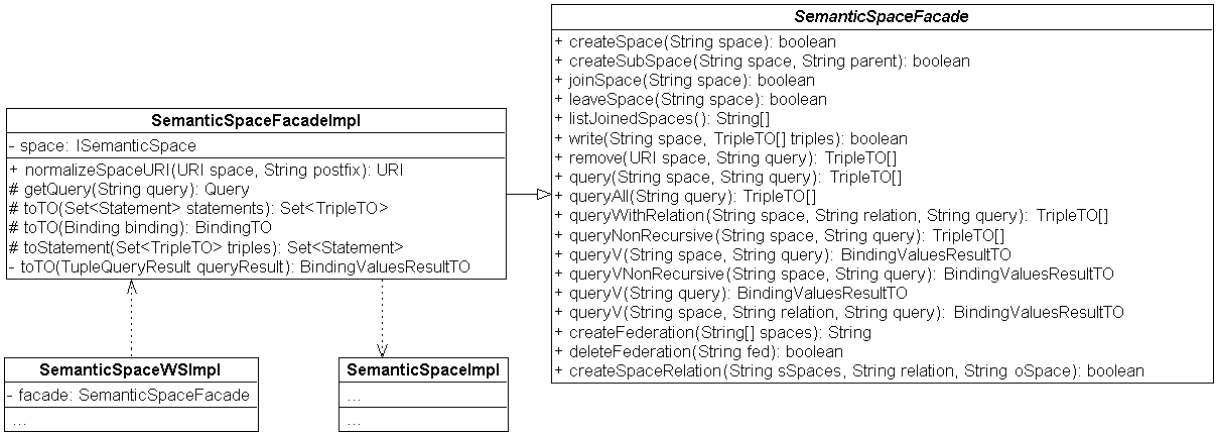

Figure 4: Space to Web service binding

3.2 Peer-to-Peer Distribution and Indexing Infrastructure

The peer-to-peer infrastructure of the semantic spaces implementation is delivered by the Overlay layer (cf. *Figure 5*). The Semanic Space Core accesses the Overlay component via the SemanticSpaceOverlayKernel class.

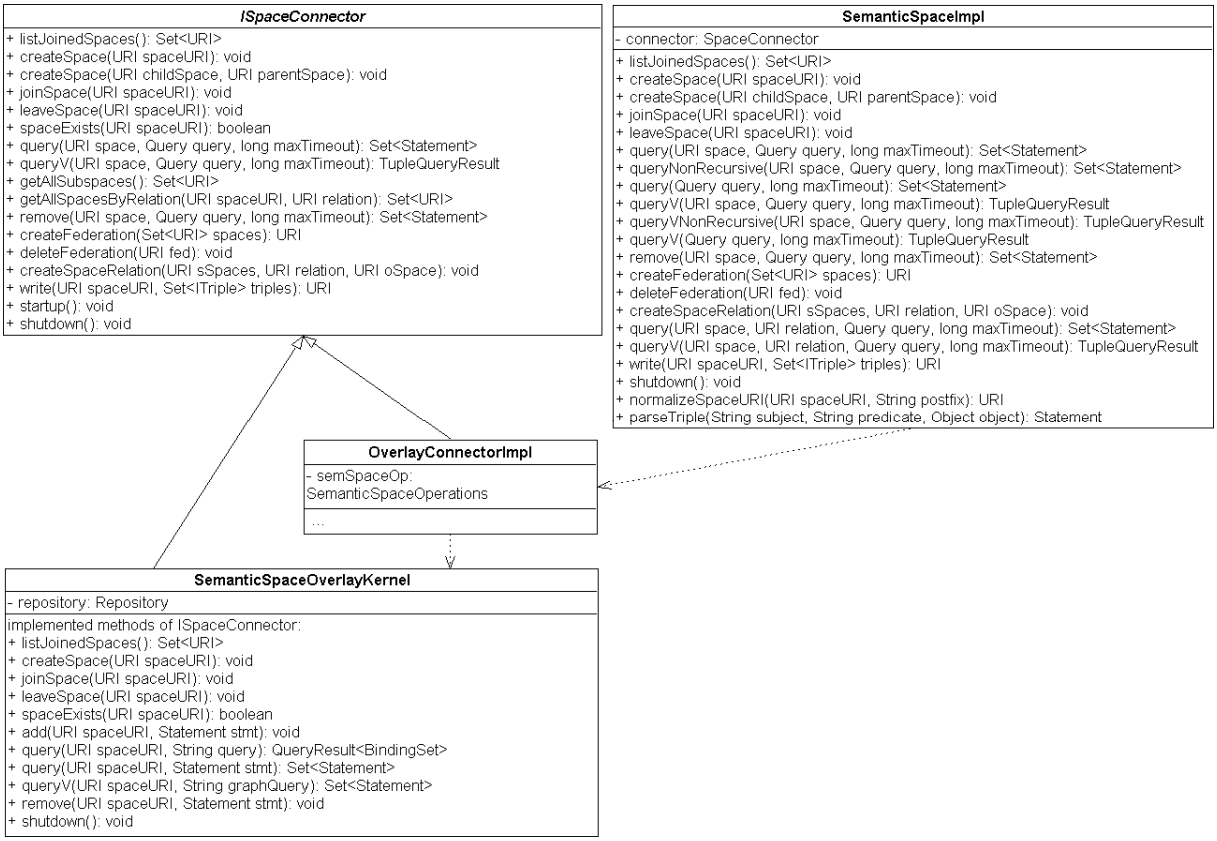

Figure 5: Space to Overlay layer

The peer-to-peer infrastructure is based on a layering of two structured overlays: CAN [4] and Chord [5]. Their implementation is established on top of the ProActive middleware that is also exploited as basis for the bus implementation and distribution. Among all existing structured overlays, the main differences lie in the way peers handle connections to other peers and how messages are routed between peers. As a fundamental component to the peer-to-peer implementation, we have developed a generic layer, which provides the common operations, found in any overlay: receiving/transmitting messages, maintaining connection to neighbors and storing/retrieving data. The specific CAN and Chord overlays respectively are implemented by adding the required code for its specific operations.

3.2.1 A generic overlay library

The basic building block is the ProActive library, which provides high-level primitives for communication and tools for deploying and debugging applications. It offers the concept of *Active Object*, a subsystem with a unique thread and its data.

A peer is implemented as an Active Object and possesses, among other things, an object of the abstract class *Overlay* that describes the overlay it is part of. In order to implement a specific overlay, it is necessary to subclass *Overlay* to provide an implementation for the following mechanisms:

- joining or leaving the overlay
- receiving and sending messages

The messages are also dependent on the overlay considered and are also specialized.

3.2.2 CAN implementation

A first component of the semantic space distribution and discovery infrastructure, as described in deliverable D1.3.2A, is a CAN overlay. The peers are organized in a 3-dimensional space corresponding respectively to the subject, predicate and object of the

maintained RDF data, and each peer is assigned a zone in the space. The coordinates are encoded using Unicode. An RDF triple is considered as a point in the 3D-space and thus, inserting a triple results in finding the peer responsible for the zone where the corresponding point lies in.

We will now describe the specific operations involved in our CAN overlay and present some architectural details.

Coordinates: As mentioned, a coordinate is an RDF triple interpreted as a Unicode triple. We use lexicographical order in each coordinate. One common operation in CAN involves finding the middle of a segment, e.g., the middle of *[a,z].* This operation is performed using a radix-2^{16} division because of the 2^{16} Unicode characters possible. The only drawback of this operation is that it sometimes gives non-printable characters but this does not affect the indexing and management of the stored data.

Bootstrapping and tracker: The first peer of the overlay, considered to be the creator of the space, assumes the control of the whole space and registers itself into a known *tracker,* specified in a configuration file. The goal of this tracker is to provide an entry point to the overlay that allows new peers to join in (Figure 6). There could be many trackers for a single overlay, for load balancing and fault tolerance purpose, as explained in Section 5.1.

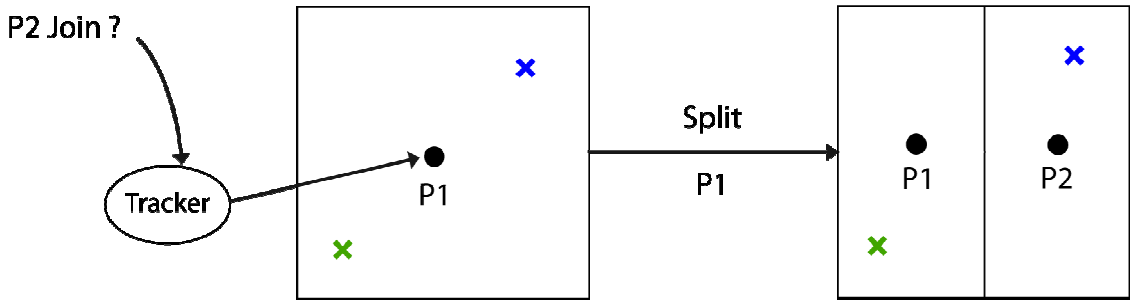

Figure 6: Example of join

Joining and splitting: When a peer wants to join an existing overlay, it contacts the *tracker*, which gives a reference to an existing peer in the space overlay. The peer sends a *Join* message and if successful, the zone will be split in two equal parts, each one under the responsibility of one of the peers. This procedure might require some data movement, as illustrated by Figure 7. The peer P2 tries to join an overlay and contacts P1. Each peer gets a new sub-zone and existing data has to be moved from peer P1 to P2. When P3 joins the network, the same process takes place but no moving of data is necessary, as all data that was maintained by peer P2 remains in P2's zone.

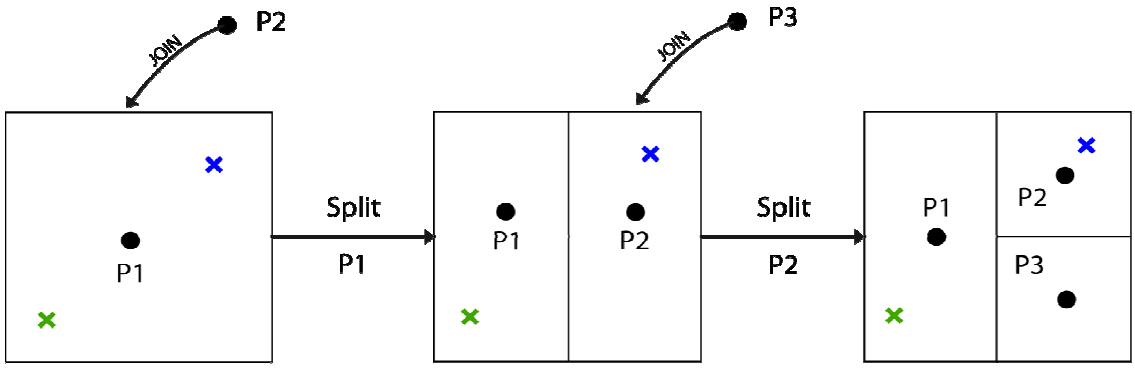

Figure 7: Successive splitting

Searching: A search is performed using a *Query* message containing the SPARQL query. When receiving such a query a peer checks whether it can be matched by data in its zone. If so, it executes the query against the local storage and returns the result to the caller. If not, it forwards the message to a neighbor along the specified coordinates; specified by bound resources in the query triple patterns.

A peer can process atomic or range queries. Conjunctives queries must first be decomposed into atomic ones by the originator peer, which eventually has to combine the results before handing them over to the requester. Figure 8 illustrates the decomposition of a conjunctive query by peer P1. The two resulting independent queries are treated separately and later, the results are merged by P1 before sending them to the caller.

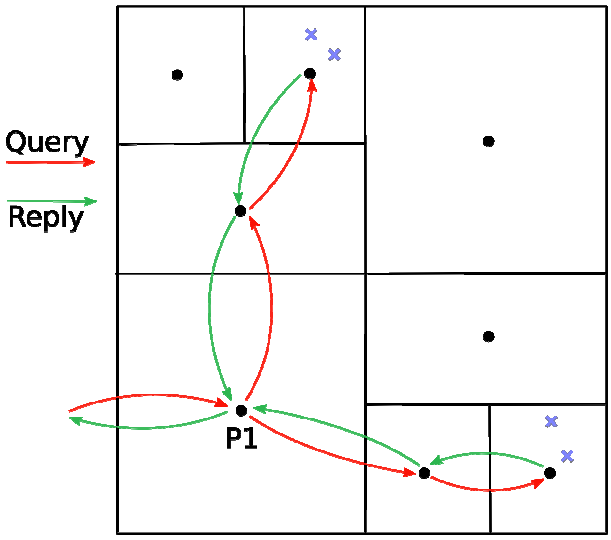

Figure 8: Example of conjunctive query

Leaving and merging: If a peer wants to leave the overlay, it chooses a neighbor and that merges the zones and accepts the data the leaving peer had stored. When the moving of the data is completed, the peer can leave the overlay. Some care has to be taken here because a peer performing a *leave* is still part of the overlay but cannot answer queries because of the moving of the data. Thus, its neighbors must hold requests until the end of the leave and then transmit them to the new zone holder. This implies synchronization with all the neighbors.

Fault handling: A space must be resilient to peer failures. The current implementation maintains the integrity of the space: if a peer leaves the network without performing the previously described protocol, its zone is taken by another one. Each peer sends periodic *heartbeat* messages to all its neighbors. If a peer does not reply within a given timeframe, it is assumed to be faulty. Among all its neighbors, one has to overtake its zone to maintain the overlay consistency. An election is started among all the potential new peers to designate the new owner and ensure a correct update of all the involved neighbor peers. Although there is no mechanism for ensuring data resilience, some form is provided by the data storage layer and the persistent storages (see Section 3.3). It is possible to have data redundancy (i.e. duplicate the data of a peer somewhere else in the space), which improves the reliability of the network, but greatly increases the complexity of the implementation. Moreover, this has an important impact on the performance.

3.2.3 Chord implementation

Chord is the second layer of the spaces infrastructure and is used to indexes spaces. In Chord, peers are organized in a ring and have a unique identifier. All data has an associated key that is used to find the peer which should store it.

Node and data identifier: Nodes and data have unique identifiers, called ID for nodes and

key for data. They are generated using the SHA-1 algorithm. Nodes are spread around an identifier ring modulo 2^m (m represents the number of bits in the identifier) ordered by the ID. The ring must be large enough to avoid hash collisions. The value m depends on how many nodes the system has to handle, but a value of 160 is large enough to maintain more than every computer in the world. Keys and node identifiers use the same number of bits. Since nodes will store references to spaces, we will use the hashed value of the space URIs as keys.

Routing information: The node located next to a node in the identifier order, is called the successor whereas the previous node is called the predecessor. For example, in Figure 9, the successor of node N8 is node N12 and its predecessor is N1. Queries can be forwarded, through each successor, until they reach the node with the closest id to the searched key.

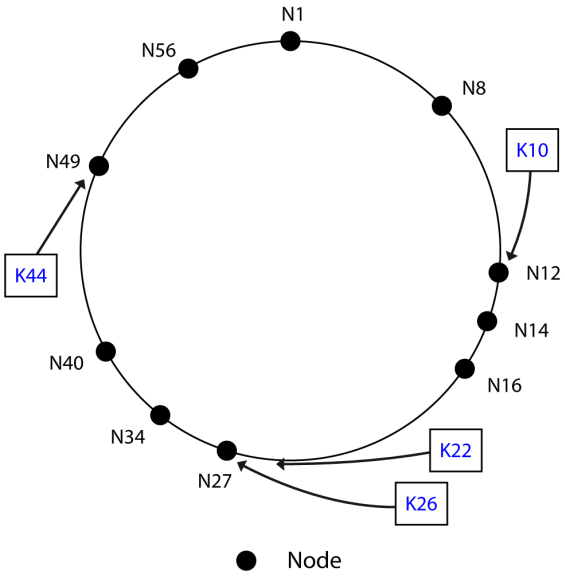

Figure 9: Example of a Chord Overlay

To improve the lookup process, Chord keeps information about nodes whose IDs are calculated exponentially and called fingers. Each node contains a finger table. Fingers are kept in a table of m entries: the ith entry in the table is defined as successor($n + 2^{i-1}$) with $1 \leq I \leq m$.

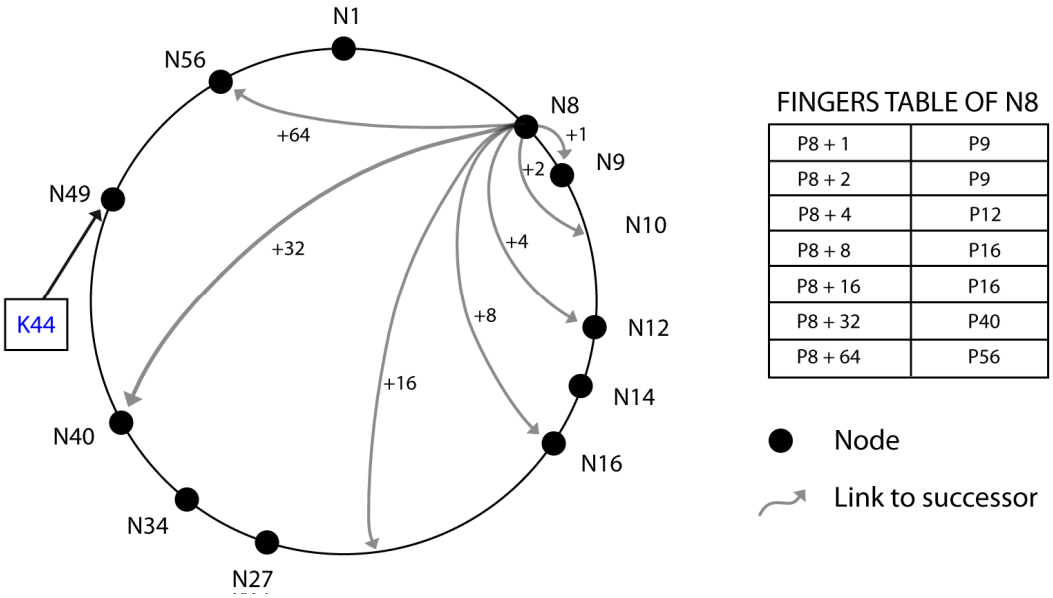

Figure 10 : Finger tables

Storing spaces in Chord: As mentioned earlier, the Chord overlay will be used to store references to spaces. There are two different ways to implement this behavior. The first one is to have a unique association between a Chord peer and a space, the key behind then equal to the peer id. This implementation is simple because the peers will not have any associated storage, as they can only store a single reference. However, the number of Chord peers will be equal to the number of spaces and although it is possible to put many peers on a single host, the overhead will be significant.

The second solution is to have a full-fledged Chord implementation and consider spaces as data which can be stored anywhere on the Chord. In this scenario, a peer can store references to multiple spaces, depending on their respective hash values. This is the approach we have chosen as it is more scalable, and has a better fault resilience.

3.3 Persistency Layer and OWLIM Bindings

Each kernel has a local repository to maintain and query the published semantic data. BigOWLIM was chosen as persistency layer because it operates with file-based indices, which allows to scale to billions of statements whereas SwiftOWLIM performs in-memory reasoning and query evaluation. Moreover, BigOWLIM is currently stronger maintained by the software provider, and also exploited by the FP7 LarKC project. This synergy with LarKC opens up means for intensified collaborations on the level of data layer progress. As the data is being stored on a physical support (ProActive-based kernels), the persistency layer allows for fault tolerance, as the data can be reinserted into the network after a kernel had crashed, provided the hardware is intact. We currently use release 3.1.a7 of BigOWLIM which is built on top of Sesame v2.2.2.

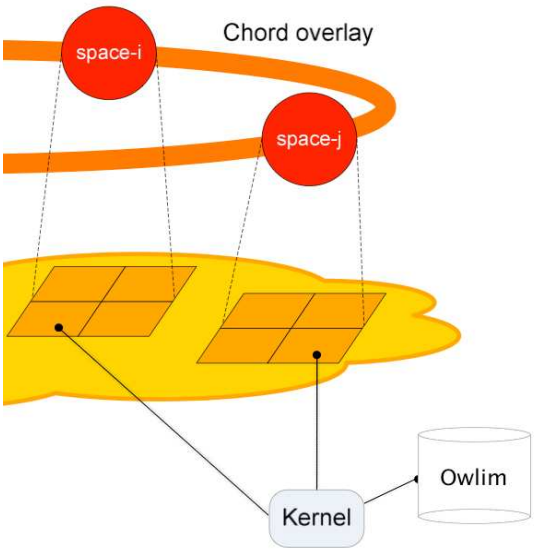

Figure 11 : Relation between kernel, space and data store

Figure 11 above shows the relationship between the kernel components, the CAN peer overlay and the local repository that is maintained by a kernel. Each peer on a CAN network has only one reference to a kernel. However, a kernel can have references to many peers that each represents a semantic space (a virtual share). When an operation is performed, for example a write operation, which aims at persisting some triples, the operation will be transferred to a space and then to a peer of this space; i.e., one of the peers that establish a CAN overlay. Finally, all data falling in the CAN zone of a given peer, is stored to the local repository of the associated kernel by a remote call. The goal of the persistency layer is thus to have a direct link to an RDF repository from any peer that shares a space.

All the operations that can be performed on the local repository associated to a kernel are implemented by the classes of the 'datastorage' package. For example, the concrete implementation for BigOWLIM is implemented in the OwlimDS class by exploiting the BigOWLIM API.

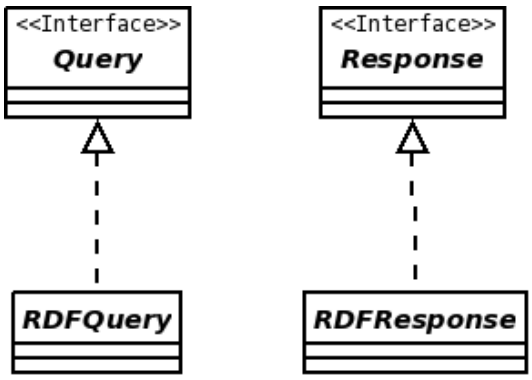

Figure 12: Query and Response interfaces

In order to access the data store, specific messages must be used. They are provided by the messages.synchronous package. Figure 12 shows the classes and interfaces used to perform a query and obtain results from a space. For example if a user wants to perform a RDF query, he will instantiate an RDFQuery and use it as parameters of the Space API. Then when the query is sent to the overlay, it will be encapsulated into internal structures and routed in the overlay.

Figure 13 shows the class diagram for all internal messages. SynchronousMessage is an

interface which abstracts the various types of queries messages that can be handled on a structured peer-to-peer network. All Internal asynchronous messages must extend the AbstractQueryMessage class which contains many common information like latency, dispatch timestamp, … for example LookupQueryMessage which is used to find a peer which manages allows finding and returns a peer which manages a particular set of coordinates (only used in intern in order to add a data on the network). , or RDFQueryMessage whichencapsulates evaluates a RDF query and returns the results foundwill trigger the creation of a Response. AbstractQueryResponse is the same for the responses. A ResponseMessage contains a QueryMessage in order to be able to be routed by using the same algorithm defines in the QueryMessage if necessary.

In addition to the store that was described above for the triple data that is published to a certain space, the persistency layer holds a separate store for maintaining information about the space logic (all spaces and the relations between them); i.e., there is a structural metadata store installed in parallel to the data storage. The two stores can be configured by means of the configuration file owlim_config.ttl. The stores are launched at starting time of the semantic space infrastructure.

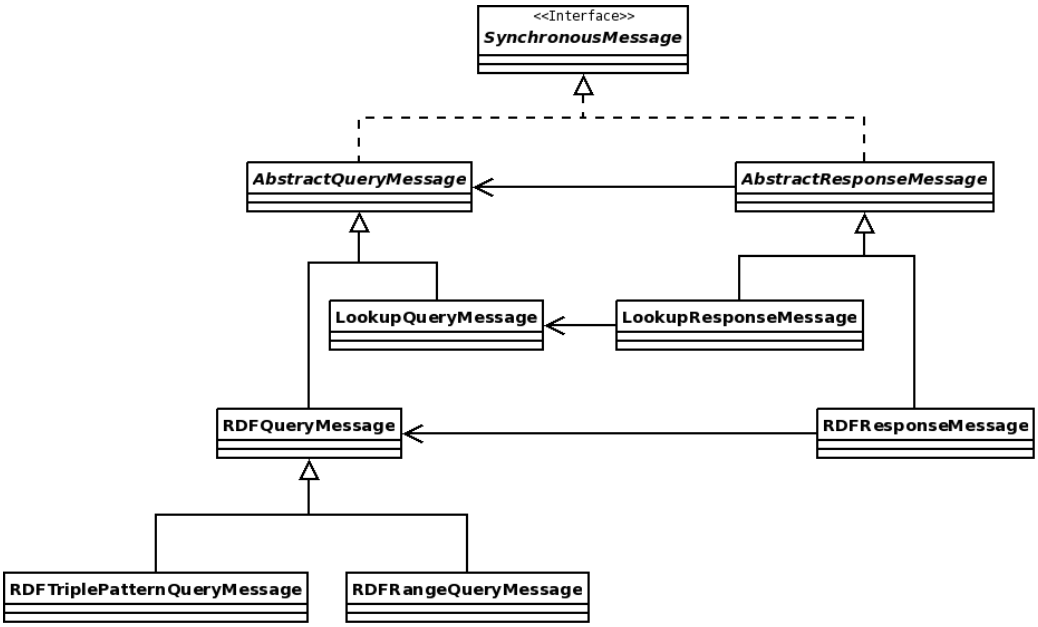

Figure 13 : Internal classes for messages in the CAN overlay

To maintain metadata information e.g. subspace logic, a metadata space is created. This space (http://eu.SOA4All/metadata) is shared by each kernel. The metadata logical rules are specified in tsontology.owl.

It is essential to call the shutdown method before terminating a kernel instance since OWLIM repositories need to be shutdown in order to be persisted. When this is not performed, a lock file in the storage folder (see Chapter 4) will prevent the repository from being restarted. This lock file must be deleted manually, in order to access the repository again.

3.3.1 tsontology.owl

The tsontology.owl is an OWL ontology and part of the semantic space implementation. It contains rules that apply to spaces and space relations. Since BigOWLIM is used as persistency layer the following rules will be inferred automatically (the domain as well as the range of the predicates are of type URI space, i.e., s0 and s1):

- s0 isSusbspaceOf s1 inverseOf s0 hasSubspace s1

- s0 hasSubspace s1 inverseOf s0 isSusbspaceOf s1
- s0 seeAlsoSpace s1
- transitive + symmetric property: s0 isSimilarTo s1
- transitive + symmetric property: s0 isRelatedTo s1

The relation s0 isSubspaceOf s1 will be automatically generated when calling the semantic space core createSpace(URI space, URI parent) method.

The relation s0 isSubspaceOf s1 will be automatically generated when calling the semantic space core createSpace(URI space, URI parent) method. Other rules in the tsontology that do not apply to spaces, are out of scope of this deliverable and will not be discussed here.

4. Installation and Configuration

The Semantic Space is accessed via WS API of the SOA4All DSB (see Chapter 3.1.2). To successfully setup the semantic space, the DSB must be installed according to D1.4.1B. In order to install and configure a semantic space, a Java Virtual Machine (version > 1.4) is required. The different layers have the following software dependencies:

- SemanicSpaceCore and ProActive both part of
    - http://www.ebmwebsourcing.com/download/SOA4ALL-DSB.zip
- BigOWLIM 3.1.0 or later
    - Since BigOWLIM is not freely available SwiftOWLIM 3.0.x or later can be also used. There are some minor differences in the configuration which will be explained later in this section. SwiftOWLIM is still in beta phase and therefore lacks some features and the full functionality of BigOWLIM cannot be guaranteed.
    - SwiftOWLIM comes bundled with all needed external libraries.
    - http://www.ontotext.com/owlim/swiftowlim-3.0.beta10-sesame-2.0.zip

The core implementation of the semantic space platform consists of the following packages and classes:

1. eu.SOA4All.dsb.space
    - ISemanticSpace, SemanticSpaceImpl, ISpaceConnector, SpaceOntology
2. eu.SOA4All.dsb.space.connect
    - OverlayConnectorImpl
    - OwlimConnectorImpl
3. eu.SOA4All.dsb.space.exceptions
    - QueryStringException, SemanticSpaceException, SpaceAlreadyExistsException, SpaceException, SpaceNotExistsException, SpaceNotJoinedException
4. eu.SOA4All.dsb.space.query
    - Query, SPARQLQuery, TriplePatternQuery
5. eu.SOA4All.dsb.space.util
    - TripleHelper

The Semantic Space Core interacts with ProActive Overlay layer through:

- eu.SOA4All.dsb.space.proactive.semanticspace
    - SemanticSpaceOverlayKernel

Some files (examples can be found in the SOA4All-dsb/SOA4All-space/SOA4All-space-impl/src/main/resources folder) must be available to the software package in order to successfully run a kernel. Those files are shortly introduced below.

The file that provides the OWL ontology that contains the semantic space logic and rules for space hierarchies and other relations: tsontology.owl file

To configure the space, an administrator has the following files at disposal. The first file is the properties file of the semantic space implementation. It allows to specify the local storage

location. The second file contains the configuration data for the OWLIM store including the specification of the ontology for the structural metadata schemes (cf. Section 3.3.1).

Semantic space configuration file space.properties (Java properties file):

- root.path: path where the OWLIM store is being created

OWLIM repository configuration file overlay_owlim_config.ttl (RDF Turtle notation):

- Contains configuration values for the OWLIM store.
- The line: owlim:imports "<PATH>/tsontology.owl"; must contain the FULL path to the tsontology.owl file.
- If using SwiftOWLIM the following changes have to be done in the configuration file: sail:sailType "owlim:Sail" has to be changed to sail:sailType "swiftowlim:Sail" and owlim:repository-type "file-repository" to owlim:repository-type "in-memory-repository";

The overlay layer which depends on ProActive, needs some specific JAVA permissions. For that reason, the following jvm property must be indicated at the startup of the jvm -Djava.policy=proactive.security.policy.

Content of proactive.security.policy;

```
grant {
    permission java.security.AllPermission;
};
```

5. Performance and Efficiency Considerations

The solutions presented in Section 3 for implementing semantic spaces rely on well-known software and architectures. However, some care has to be taken to provide good performance and scalability. In this section we will present the mechanisms that have been implemented to support increasing numbers of peers and queries.

5.1 Tracker scalability and resilience

The tracker maintains a list of remote peers. However, for scalability reasons, a tracker cannot maintain a reference to all peers in the overlay. Our first optimization consists in having only a subset of the peers registering with the tracker. For that we use a probability rate, which is by default fixed to 0.2 but it can easily be modified. A second optimization consists in having the tracker return a randomly chosen peer, so that new peers do not always contact the same one. The overall space management is thus better distributed. Finally, to improve fault tolerance, it is possible to use multiple trackers for the same overlay.

5.2 Optimized merge operation

When leaving the overlay, a peer must merge its zone with one of its neighbors and transmit its data. The algorithm presented in the original CAN paper [4] is recursive and might entail many data movements.

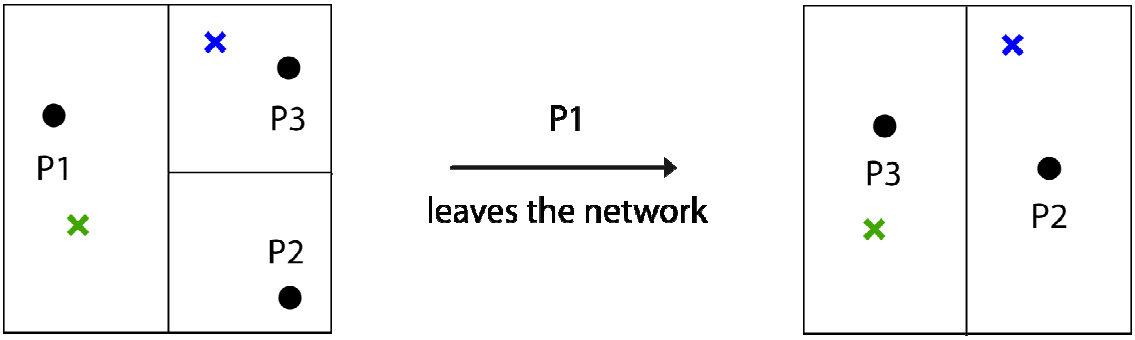

Figure 14 : Example of merging without optimization

The left side of Figure 14 shows a 2D-CAN overlay with three peers and two data stored data items (crosses). When P1 decides to leave the network, the default algorithm produces the new repartitioning on the right. As we can see, the zone managed by the remaining peers has changed and since data is associated to a zone, they have to be moved between peers. The data previously owned by P3 (respectively peer P1) is now owned by P2 (respectively P3). This requires at least two copying procedures but there might be further intermediate data moving required. This clearly slows down the *leave* operation and puts a heavy load on the network.

We have modified the algorithm to remove the recursive part and limit the data movement. To do so, each peer has to maintain a history record of all splitting and merging involving its zone. When merging, a peer simply revert the last split done. This adds a negligible memory overhead but greatly improves the performance.

Figure 15 shows the result of the modified *merge.* To achieve this result, a single moving of data is required: a half from peer P1 to P2 and the other half from peer P1 to P3.

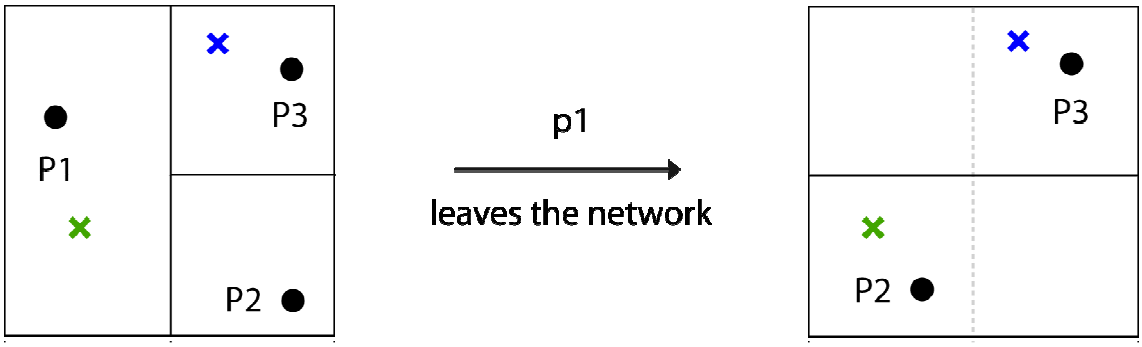

Figure 15: Example of optimized merging

5.3 Imbalance in data storage

The selection of a zone for storing data is base on the lexicographical order, as stated in the previous section. Consequently, the longer a common prefix between two different pieces of data, the more likely they will belong to the same zone and thus be stored on the same peer. This can lead to an imbalance for which some peers have to handle a lot of data whereas some have none at all, as shown in Figure 16. We consider a 2D-CAN where all data is prefixed by some closely related letters on the x-axis. Although there are two zone of roughly equal size, all the data is stored in the first one only.

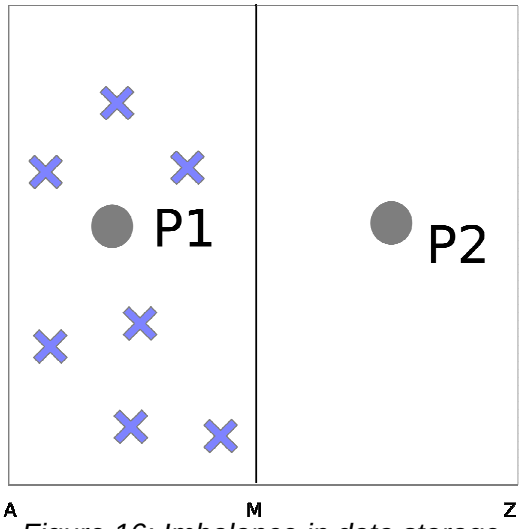

Figure 16: Imbalance in data storage

First, although we do not have any control over the semantic data to be stored as RDF triples, BigOWLIM enforces the use of URIs that start with a scheme field (*http, ftp...*). To avoid having clusters of data starting with the same scheme, we ignore it and consider only the remaining of the URI when computing the storage zone. This is done transparently and does not modify the data, but only the indexing.

Second, to further avoid data clustering, it is possible to modify the way a zone is split. By default, a split will generate two zones of equal size (for each dimension it is split in the middle). Instead, we can split a zone such that the volume of stored data is the same in the two resulting sub-zones.

Finally, it is possible to incorporate load-balancing mechanisms at runtime, which allows peers to modify their zone to increase or decrease the volume of data they store. Our current

implementation does not support this feature but it envisaged to be added for the next major release, if necessary.

5.4 Asynchronous processing of queries

Processing a query might require the collaboration of multiple peers for routing or searching in the data stores. Although the *Active Object* model forces single threaded peers, we use asynchronous communications to maintain better performance. A peer can thus forward a query, perform some other operations, and receive a reply later. To do so, each peer maintains a list of forwarded requests awaiting a reply, with a timeout to handle faults. When a reply reaches a peer, it forwards it to the sender of the initial query If no reply is received after a given timeout, the caller is notified of the problem by means of a dedicated warning message.

6. Conclusions

This deliverable described the first semantic space implementation. The implementation delivers the space logic, P2P-overlay-based indexing for distribution and discovery as they were specified in Deliverable D1.3.2A. The implementation is deployable on top of ProActive, and provides the space-specific parts of the SOA4All Distributed Service Bus. The semantic space prototype thus complements the bus prototype that is shipped with deliverable D1.4.1B SOA4All Runtime.

The main focus of the deliverable was in presenting the prototype. This includes information on how to install, configure and run the semantic space platform. Furthermore, the document provides detailed descriptions of the different components of the prototype in order to better understand the work done and the functionality provided. The three main components are the space logic with the API implementation, the P2P-based indexing infrastructure and the OWLIM storage and query engine bindings.

In addition to the core aspect of the deliverable, in Section 2, we re-visited the specification and elaborated on two concerns that were raised by the reviewers; namely, redundancy, and the update functionality. Moreover, this deliverable also provided some more detailed considerations in regards to performance and scalability of the semantic space platform. This was another issue that was insufficiently covered by the original specification in Deliverable D1.3.2A. First evaluation results confirming these improvements are expected to be delivered by month M24.

7. References

1. Reto Krummenacher, Imen Filali, Fabrice Huet, and Francoise Baude: Distributed Semantic Spaces: A Scalable Approach To Coordination. SOA4All project deliverable D1.3.2A, March 2009.
2. Reto Krummenacher, Ioan Toma, Christophe Hamerling, Jean-Pierre Lorre, Francoise Baude, Virginie Legrand, Philippe Merle, Cristian Ruz, Carlos Pedrinaci, Dong Liu, and Tomas Pariente Lobo: SOA4All Reference Architecture Specification. SOA4All project deliverable D1.4.1A, March 2009.
3. Christophe Hamerling, Virginie Legrand, Francoise Baude, Elton Mathias, Cristian Ruz, Michael Fried, and Reto Krummenacher: SOA4All Runtime. SOA4All project deliverable D1.4.1B, August 2009.
4. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, 2001: 161-172.
5. Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan: Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. IEEE/ACM Transaction on Networking 11(1), 2003: 17-32.

