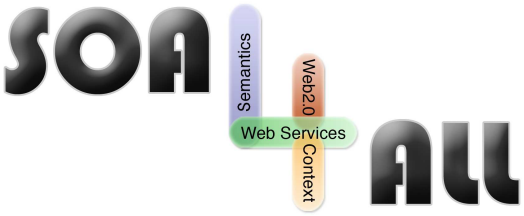

Project Number: **215219**

Project Acronym: **SOA4All**

Project Title: **Service Oriented Architectures for All**

Instrument: **Integrated Project**

Thematic Priority: **Information and Communication Technologies**

D1.3.3A A Distributed Semantic Marketplace

Activity N:	Activity 1: Fundamentals and Integration Activity
Work Package:	WP1: SOA4All Runtime
Due Date:	28/02/2010
Submission Date:	28/02/2010
Start Date of Project:	01/03/2008
Duration of Project:	36 Months
Organisation Responsible of Deliverable:	UIBK
Revision:	1.0
Author(s):	Reto Krummenacher (UIBK), Fabrice Huet (INRIA), Michael Fried (UIBK), Laurent Pellegrino (INRIA), Ivan Peikov (Ontotext), Alex Simov (Ontotext)
Reviewer(s):	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**
PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	

CO	Confidential, only for members of the consortium (including the Commission)	

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	2010-01-28	Initial TOC	Reto Krummenacher (UIBK)
0.2	2010-02-10	Semantic Space 2.0 Section	Michael Fried (UIBK)
0.3	2010-02-16	Section 2 Section 3.3/3.4 Section 3.5	Fabrice Huet (INRIA) Fabrice Huet Ivan Peikov (ontotext)
0.4	2010-02-19	Final Draft (Internal Review)	Reto Krummenacher, Fabrice Huet
0.5	2010-02-23	Incorporation review by Dong Liu	Reto Krummenacher
0.6	2010-10-25	Incorporation review by Yosu Gorronogoitia	Reto Krummenacher, Michael Fried, Fabrice Huet
1.0	2010-10-25	Final release for submission	Reto Krummenacher

Table of Contents

List of Figures

List of Tables

Glossary of Acronyms

Acronym	Definition
API	Application Programming Interface
BC	Binding Component
BPEL	Business Process Execution Language
CAN	Content Addressable Network
D	Deliverable
DSB	Distributed Service Bus
EC	European Commission
EJB	Enterprise Java Beans
ESB	Enterprise Service Bus
FP	Framework Program
FP7	The 7th Framework Program
HTTP	HyperText Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineerings
M	Median, Milestone
OWL	Web Ontology Language
P2P	Peer-to-Peer
RDF	Resource Description Framework
SE	Service Engine
SOA	Service-Oriented Architecture
SOA4All	Service-Oriented Architectures for All
SOAP	Simple Object Access Protocol
T	Task
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WP	Work Package
WSMX	Web Service Execution Environment

© SOA4All consortium

Executive Summary

The core motivation for semantic spaces is to add an additional dimension to the communication possibilities of the SOA4All service bus infrastructure. Semantic spaces enable a 'publish and read'-style interaction mechanism for services, but also for monitoring probes, user management services and various repositories and platform services. These original objectives were governing the original semantic space specification, and have given the approach a very service communication-driven touch. The recent emergence of the Linked Data initiative (www.linkeddata.org) and the advent of more mash-up-style, and hence data-driven service marketplaces, have led to a focus on RDF data sharing and interlinking. For this purpose, the updated semantic spaces infrastructure brings the space idea closer to linked data and offers now more standards-compliant RDF processing and storage.

The goal of this deliverable is thus to enhance and improve the specification of the SOA4All semantic space infrastructure in terms of interfaces, linked data compatibility and RDF data-driven mash-up realizations. The implementation of these extended ideas is ongoing work and will be released latest with the M30 milestone.

The deliverable first presents a short recapitulation of the main ideas around semantic spaces in SOA4All and state some of the principle decisions that were taken in the project. The focus is then set on the updated architecture and interfaces, some discussion and guidelines of particular specialized implementations in terms of overlay management, notification services and extensions towards clustered realizations. Finally, the deliverable provides a concrete example of a (simple) distributed marketplace. This data-driven approach to process execution is based on the consumption and production of RDF that is shared in semantic spaces.

1. Introduction

From the very beginning, semantic spaces were intended to add an additional dimension of communication to the service bus infrastructure of SOA4All. Through semantic spaces the SOA4All Runtime can offer a 'publish and read'-style interaction mechanism for services, but also for monitoring probes, user management services and the various repositories and platform services. All of these original objectives of semantic spaces are very service communication-driven. The recent emergence of the Linked Data initiative (www.linkeddata.org) and the advent of more mash-up-style and hence data-driven service marketplaces led to a focus on RDF data sharing and interlinking. For this purpose, the second generation semantic spaces in SOA4All, termed Semantic Spaces 2.0, intend to bring the space idea closer to linked data and to serve as a standards-compatible RDF manipulation and storage infrastructure. The primary intention is to evolve semantic space technology towards a multi-purpose and thus highly horizontal infrastructure for the realization of distributed and dynamic marketplaces of semantic data.

Such marketplaces are helpful in the context of manipulating monitoring data, where low-level probes collect raw data about the SOA4All Runtime and service and process execution, where aggregators select and filter raw data to infer higher-level monitoring information that is presented and further processed by users via the SOA4All Studio. The management of monitoring data is thus a first example of a highly collaborative and interactive application scenario of semantic spaces. The advantage of having the spaces being made more compatible to standard RDF storage interfaces (in the case of Semantic Space 2.0 we inherit the RDF2Go framework, see Section 3) is its easier adoption. Spaces can now more simply be used in applications that so far relied on single-node RDF repositories, without requiring major re-engineering and programming. This was considered important in the context of SOA4All, as many components and platform services rely on the sharing, and hence storing, of semantic artefacts in RDF. While the first specification was still focusing mainly on traditional tuplespace and triplespace assets, such as the parallelization of processes and Web service executions, the Semantic Space 2.0 specification more carefully responds to the needs of large scale semantic data management scenarios; without neglecting its original intentions, of course.

A promising service coordination-focused application scenario of the update semantic spaces infrastructure is the execution of Semantic Web service mash-ups that fully rely on the consumption and production of semantic data. In this document we present an approach to service coordination over semantic spaces. Activities of a process read the required data from a process space and write the result of the service execution back to the process space for further use by services that are invoked later in the workflow. An important aspect of this proposal is the fact that activities are operating with RDF data only, and invocation and production are entirely SPARQL-minded.

This RDF and SPARQL-driven approach to the establishment of light-weight service economies emphasizes again the need for changes to the interaction model of semantic spaces towards more standardized models for RDF and RDF querying. Still, even so we adopt some new RDF2Go operations, the core principles of subspaces and federations that allow for dynamic and distributed structures of spaces are not changed. These means of structuring spaces are important for maintaining scalability and flexibility in communication and coordination of various semantic artefacts.

The goal of this deliverable is thus to enhance and improve the specification of the SOA4All semantic space infrastructure in terms of interfaces, linked data compatibility and mash-up realizations. The implementation of this extended ideas are ongoing work and will be released latest with the M30 milestone. The software will be delivered as Deliverable D1.3.3B. As such this deliverable provides guidelines and first details for and about the upcoming implementation tasks. Moreover, it presents first ideas and implementation ideas

about how to execute processes based on purely data-driven service mash-ups, as stated in the previous section.

In Section 2 we present a short recapitulation of the main ideas around semantic spaces in SOA4All and state some of the principle decisions that were taken in the project. Section 3 presents the specification of the Semantic Spaces 2.0 infrastructure. We start with an updated architecture overview and then enter in more details about the RDF2Go-compatible interfaces, an overview of the specialized implementations, and some first pointers towards an updated notification services realization. Semantic Space 2.0 resolves subscriptions at the level of the underlying data stores, and listeners installed at the level of the data stores trigger the desired notifications. In Section 4, we summarize the work done around the realization of distributed marketplaces. We present a data-driven approach to process execution that is based on the consumption and production of RDF that is shared in semantic spaces. With Section 5 we conclude this deliverable.

2. Recapitulation of Semantic Space Principles and Use

In SOA4All, Semantic Spaces are used for two main purposes: i) for storing semantic artefacts, and ii) for asynchronous, publication-based communication between Distributed Service Bus nodes and platform services (see Deliverable D1.4.1A [1]). In the case of storage, the Semantic Space infrastructure is primarily used as distributed repository for semantic description of services (SOA4All platform services, but also published business services), and as shared memory for RDF-based contextual data and monitoring data that is gathered at the level of the service bus.

The Semantic Space infrastructure provides a blackboard or shared memory-based storage, communication and coordination platform for SOA4All platform services that rely on the exchange and persistency of RDF-based semantic data. Moreover, the functionality of the Semantic Space is also offered to users and business services via respective Binding Components that are exposed by the Distributed Service Bus.

The main requirements for the SOA4All Semantic Space infrastructure are distribution, openness and scalability. In order to provide a fully distributed, globally accessible and scalable approach to Semantic Spaces, it is crucial to enable means that also distribute the data of individual spaces. Otherwise, we are limited in size and dependability to the limits of individual server nodes called kernel. As a consequence of this, SOA4All exploits peer-to-peer technology to distribute the triples that are published and stored in the Semantic Space infrastructure. The P2P semantic space infrastructure is based on a mix of two structured overlay networks, CAN and Chord (Figure 1). Further details on the architecture and the implementation can be found in deliverable D1.3.2B [3].

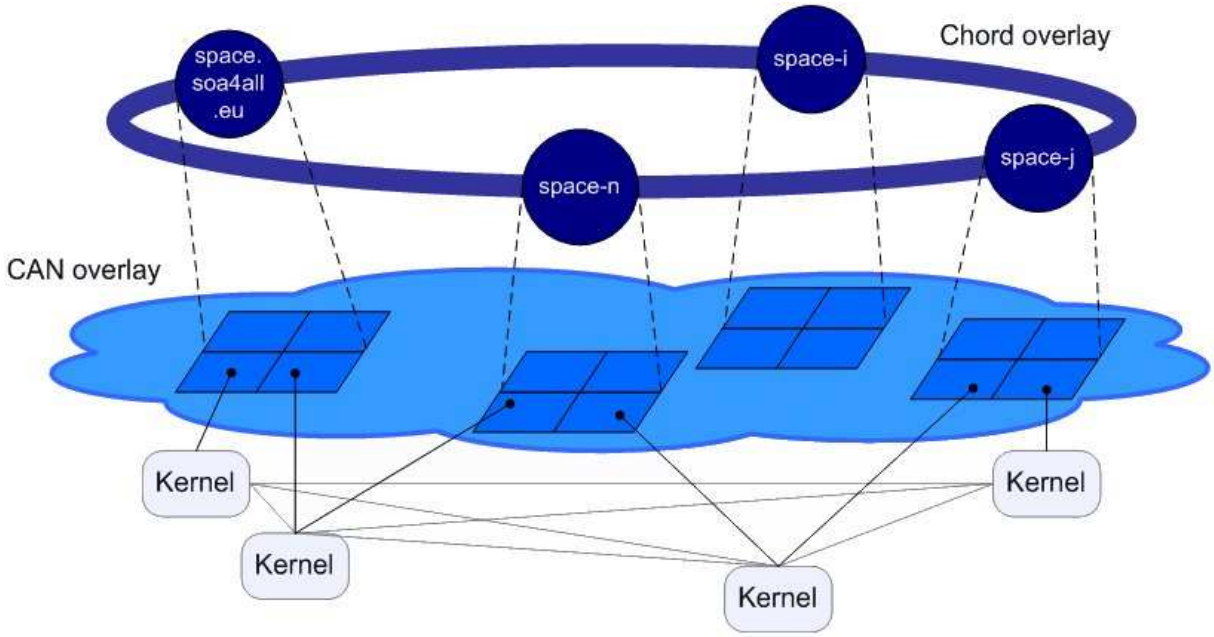

Figure 1: P2P overlay structure.

The data model supported by the SOA4All Semantic Space infrastructure is the Resource Description Framework (RDF). RDF is the de facto standard for the representation and formalization of information about resources in the World Wide Web, and is particularly intended to be used for the representation of metadata (in the case of SOA4All: monitoring data, service and process description or user profiles).

3. Semantic Spaces 2.0

The second version of semantic spaces aims at offering a more standardised RDF/SPARQL interface than its first specification did (Deliverable D1.3.2A [2]). The second version follows the interface definitions in the RDF2Go (http://rdf2go.semweb4j.org) framework and extends it with the introduced structural features of spaces such as subspace hierarchies and federations. RDF2Go is an abstraction over triple (and quad) stores, which allows developers to program against RDF2Go interfaces and choose/change the underlying implementation later more easily. Currently there are adapters available for Sesame/OWLIM and Jena. This second and final release of the spaces infrastructure that follows with the M30 milestone will then be accessible via a standardized SPARQL endpoint, via a WSDL/REST service specification and, as initially proposed, as a native Java software package.

3.1 Architecture

In contrast to the first release of semantic space, the new implementation achieves a more standardized interface (with regard to SPARQL/RDF compatibility and general triple store implementation patterns instead of a custom write/query logic) based on RDF2Go methods. By adding space specific methods (e.g.: createSpace(URI space), deleteFederation(URI federation)) and adding a space parameter to existing RDF2Go methods (e.g. sparqlConstruct(URI space, String query) instead of sparqlConstruct(String query) or add(URI space, Statement statement) instead of add(Statement statement)), the system is extended to follow the space paradigm. The differences from Semantic Spaces 1.0 as presented in [3] will be explained in detail in the next paragraphs. The implementation uses an AbstractModelSet to enable different models to be attached. Figure 2 shows an overview of the overall architecture. The new semantic space is exposed via WSDL/Rest WS and the SPARQL query functionality can be accessed over a standard compliant SPARQL endpoint. The space can be integrated in a native application as well.

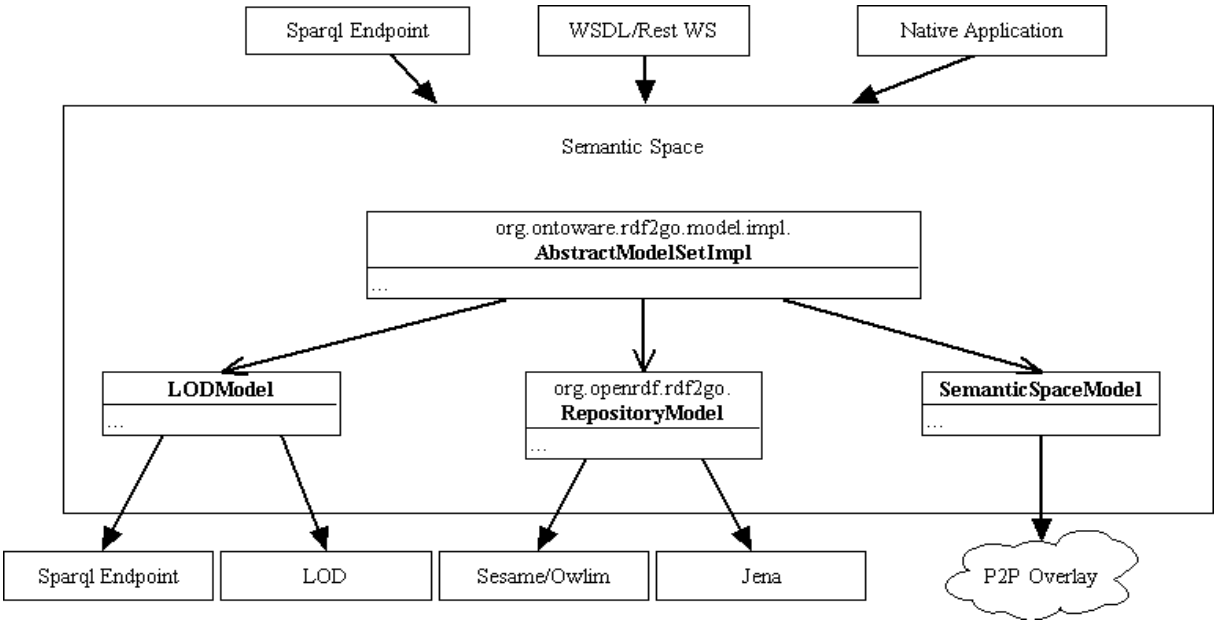

Figure 2: Overview of the Semantic Space 2.0 architecture.

The space offers the following operation models:

- LOD Model: Enables to query data from any SPARQL endpoint and the Linked Open Data cloud. Therefore, the LOD cloud becomes part of the space and vice-versa.

Especially for the execution of queries, it is important to introduce a scope (limiting results) to improve search speed.

- Repository model: use the space as replacement for a regular triple store. Since RDF2Go as a top layer wraps the actual triple stores the choice of the underlying implementation can be changed easily. In this model, the space consists of a single machine, which naturally limits the amount of information that can be stored. Since distribution is missing, the space is also dependent on a single server or WS entry/endpoint.
- Semantic Space model: Enables distribution of the space over various machines. The SemanticSpaceModel inherits the AbstractModelSetImpl and can be attached to the space instead of the LODModel or the RepositoryModel. The distribution is a crucial point in the space paradigm. RDF2Go provides the possibility to use different triple store implementations on different space nodes by adding a layer in between the distribution/space logic. Similar to the LOD model there has to be a focus on query optimisation concerning query distribution and return of the results.

3.2 Interfaces and RDF/SPARQL Compatibility

This paragraph introduces the Semantic Space 2.0 interfaces and discusses the differences from the previous version of the space. Figure 2 shows the new ISemanticSpace interface, which has been split into three sub-interfaces that represent different types of functionalities that need to be identified at the level of the P2P Overlay. Indeed, ISemanticSpaceManagement contains all operations that can alter the overlay structure. These operations have some impact on the time complexity of routing of the overlay. Only administrators need to execute them. However, ISemanticSpaceOperations and ISemanticSpaceNotification contain operations that are performed on the overlay data stores. These operations have no impact on the overlay maintenance and can be performed by all users. In comparison to the first semantic space implementation that was released with [3], the new one offers (almost) full SPARQL compatibility, except for the DESCRIBE queries. The old very restricted (with regard to SPARQL compatibility) query and triple pattern logic have been removed completely. The space accepts triples in multiple notations and returns RDF/XML when executing CONSTRUCT queries and a SPARQL protocol-compatible XML serialization when invoking SELECT queries [4]. ASK queries are answered with a Boolean value.

The ISemanticSpaceManagement interface contains operations to handle the creation of spaces and subspaces. The ISemanticSpaceNotification interface enables the publish/subscribe functionality of the space. Both interfaces contain functionality also included in the old space implementation. Most methods of the ISemanticSpaceOperations interface have been inherited from RDF2Go, except the operations for joining, leaving and the listing of spaces, the handling of federations and space relation, and the registering of existing (atomic) SPARQL endpoints. All other RDF-related operations have been derived from the RDF2Go Model interface and were extended with a parameter to indicate the identifier of the space to operate against. This limits the execution scope of the operations to the given space, instead of the whole storage environment. The following operations of the original semantic space specification have been replaced with RDF2Go-compatible operations:

- write: the new space implementation offers various add methods instead of just one. This enables to write single/multiple statements at once and contain explicit datatype and language tag support, as they are given by the RDF2Go framework.
- remove: triples are removed analogue to the add methods. Unlike the first space implementation the deleted triples are not returned but deleted silently.

- query: Instead of the query and queryV (nonrecursive, etc.) methods the new space offers almost full SPARQL compatibility (with exception of describe, which has not been implement). In addition, a find method (using triple patterns) and various contains methods (map to SPARQL ASK queries) complete the query functionality of the new space implementation.

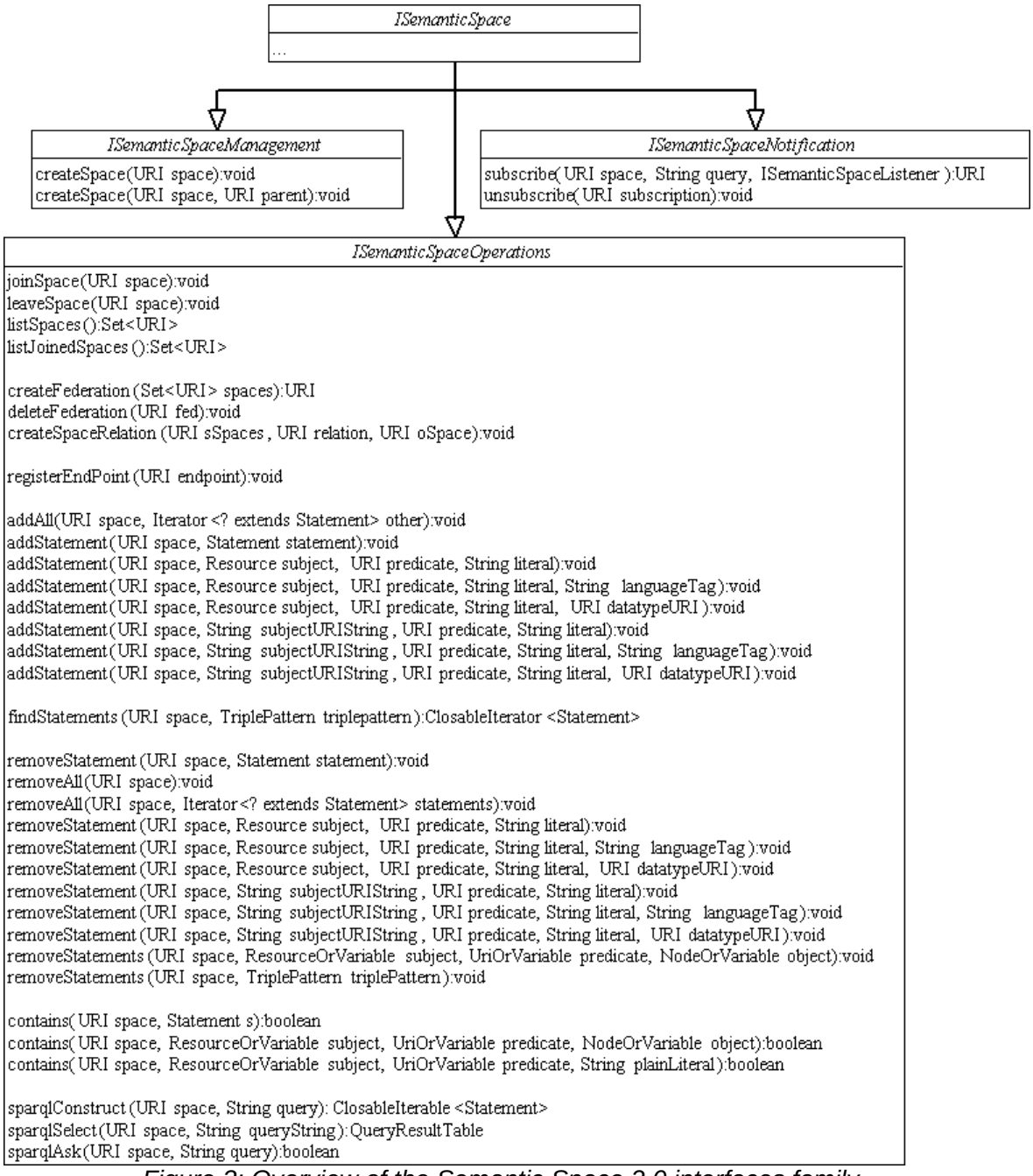

Figure 3: Overview of the Semantic Space 2.0 interfaces family

The following additional space related methods have been added to the new specification:

- listSpaces: lists all existing spaces in addition to the listJoinedSpaces method. This method was missing in the old version but is important to evaluate which spaces exist and can be joined.

- registerEndpoint: registers a public SPARQL endpoint to query it like a regular space. Note that in comparison to spaces there is no write/remove functionality available for such endpoints. The method has been introduced to access endpoints directly instead of copying the data to a space and then querying it.

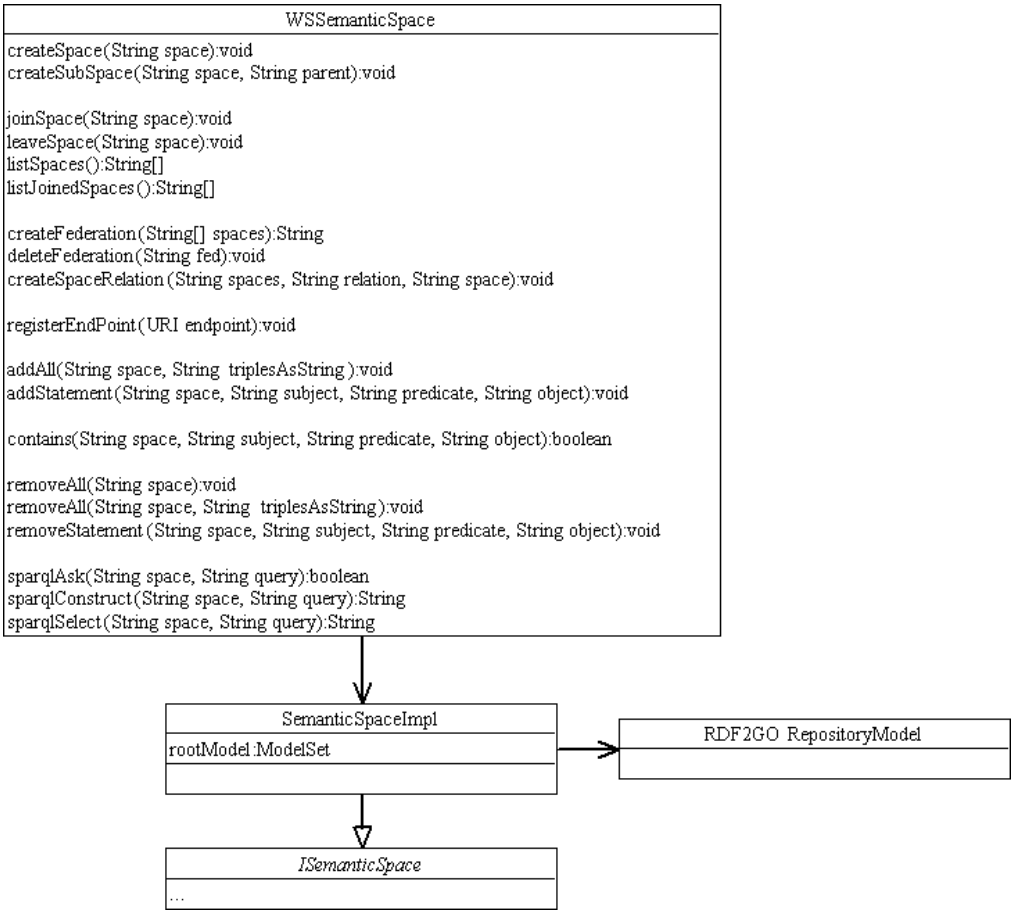

Figure 4: Semantic space access via Web Service

To simplify the access, the WS interface of the semantic space shown in Figure 4 does not expose all methods of ISemanticSpace, since most add/remove/contains methods can be substituted by one or two methods. The WS implementation uses standard datatypes (e.g., String) as parameters and return values. The following operations can be executed:

- Space methods to create, join, leave and list spaces. Additionally the federations logic, createSpaceRelation as well as the registerEndpoint method can be accessed.
- Two add methods:
  - *addAll*(String space, String triplesAsString): the triples have to be provided as a String in one of the following notations: RDF/XML, Ntriples, Trig, Trix or Turtle
  - *addStatement*(String space, String subject, String predicate, String object):
- Three remove methods
  - *removeAll*(String space): remove all triples form a space
  - *removeAll*(String space, String triplesAsString) and *removeStatement*(String space, String subject, String predicate, String object): analogue to the add methods.

- Query possibilities via *contains* (returns true if the query is satisfied), *sparqlAsk* (returns true if the query is satisfied), *sparqlConstruct* (returns the result as RDF/XML) and *sparqlSelect* (returns a result SPARQL XML protocol serialization). In fact, the *contains* operation is a short-cut operation provided by RDF2Go to simulate an ASK query for a specific triple.

3.3 P2P Overlay Architecture

The introduction of the new interface implies updating the existing implementation. First, as several actions associated to the previous API are encapsulated in messages, their signature must be updated according to the new RDF2Go API. In the same way, after a query, several responses can be returned and need to be merged. Depending on the type of the RDF2Go interface, the merge must be done appropriately.

Compared to the previous version, the RDF2Go interfaces introduce new methods that have to be handled as messages at the overlay level. Several of them require a similar routing. This point involves introducing a notion of router that can be reused and specialized for each method. As seen in the previous section the query and queryV requests that needed to be routed on the overlay have been replaced by specific SPARQL queries (ask, construct, select). The dispatching of the new queries is now performed by a query manager in order to abstract the action to execute (identification, creation of messages from public/private API…), depending of type of the query to handle.

The last item regarding the architecture is about operations entry point. In the previous implementation, all operations were performed directly on the *kernel* (SemanticOverlayKernel). Now, its use is restricted to administrative operations, which alter the overlay structure. Trackers have been chosen as new entry point so that applications do not need to know references to *kernels* in addition to the Tracker's references.

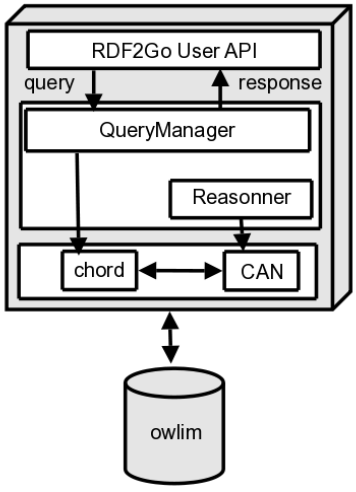

Figure 5: The new P2P Overlay Architecture

In terms of implementation, each Tracker implements the RDF2Go interface and delegates most operations to peers. For this reason, each peer is encapsulated in a SemanticPeer also implementing the new RDF2Go interface. Naturally, the routing of operations differs from Chord or CAN peer: at the Chord level, an operation must be routed toward the specified space whereas, at the CAN level, the operation is routed towards peers storing the requested data.

Figure 5 shows the new P2P Overlay architecture. At the top, a user can perform an

operation using the RDF2Go API from a Peer or a SemanticOverlayKernel. If the operation is sent from a Chord Peer or a SemanticOverlayKernel, the goal is to get a CAN peer from which we can route the query to the correct datastore(s). In order to perform this correct routing, the query is transmitted to a QueryManager which has three objectives. The first one is to identify the query type from the public API and to build an equivalent set of queries from the private API by using the Reasonner. The second goal is to dispatch the queries and to wait for responses. At this level, the queries will be routed step by step in the overlay. Finally, the third objective is to aggregate response(s) received and to construct a response which will be returned to the user.

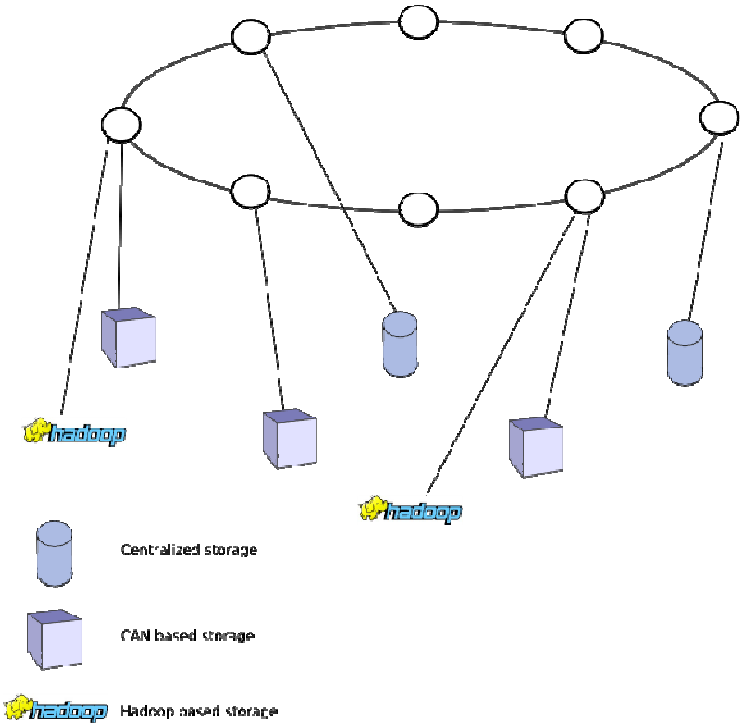

Figure 6: Multi-implementation-base semantic space architecture

3.4 Space Specialization

Following discussions and comments during the previous evaluations, we came to the conclusion that different versions of the Semantic Space were needed, depending on the application requirements. The CAN version can be deployed on a large-scale network and encompass a large number of machines, managing a very large number of triples. However, this has an impact on the performance, due to the network communications involved with complex queries. Some applications might, on the other hand, manage a small number of triples but have high performance requirements. We have thus decided to provide different implementations of the Semantic Space, which can be installed depending on the user needs.

- A centralized version (all triples in a single repository), accessible through a Web service or directly through the RDF2Go-compliant API
- A distributed and large scale version based on the CAN overlay implementation

Moreover, we are currently investigating a Hadoop-based implementation [5], which could lead to very high performance on a cluster.

It is important to note that, thanks to the RDF2Go API, all these versions of the Semantic Space can be accessed through the exact same API, making them transparent to the user.

They can also be combined at the Chord level to federate them, as detailed in previous deliverables. Overall, we envision the architecture shown in Figure 6, which should be flexible enough to suit the various needs of semantic data producers and consumers while maintaining a unique and global Semantic Space.

3.5 Notification Services

Notification services are provided by the framework via the ISemanticSpaceNotifcation interface. A client of that interface subscribes for notifications for incoming statements matching certain graph pattern. The notifications are asynchronous and are implemented according to the observer design pattern. The subscriber provides a ISemanticSpaceListener instance with the following notification callback method:

notifySubscriber(URI subscription, Statement matchingStatement)

Further on, whenever the storage subsystem sees a statement matching the graph pattern defined by this subscription the notifySubscriber method of the client-provided listener is called, provisioned with the subscription identifier URI and the matching statement itself.

The unsubscribe operation is also supported and it cancels a certain subscription identified by its URI.

A possible extension of the ISemanticSpaceListener interface would be a method to be called back with all the query solutions (binding sets):

notifySubscriber(URI subscription, Map<String,Value> querySolution)

This extension will be considered in future versions of the notification services and is not currently supported.

In order to define the graph pattern of interest the client of the notification services provides a SPARQL SELECT query as a plain string. It should be noted though that such a representation is meant for simplicity and future extensions compatibility rather than to imply full SPARQL support. The SPARQL query is expanded into a set of graph patterns which are then used to filter incoming statements and notify the subscriber about those of them that help form a new solution of at least one of the graph patterns. At the current stage, more complicated SPARQL constructs like FILTER, OPTIONAL, DISTINCT, LIMIT, ORDER BY, are to be ignored by the notification service. However, the somewhat simplified query subscription interface allows for further extensions of the graph pattern matching functionality without breaking existing clients (e.g. FILTER support might be seamlessly added in future versions without changing the subscription interface just by extending the interpretation of the subscription query).

By removing the LIMIT, DISTINCT and ORDER BY from the graph matching implementation (and also from the distributed nature of the storage service) we imply that the subscriber should not rely on any particular order or distinctness of the statement notifications to follow his subscription. Duplicate statements might be delivered in response to a graph pattern subscription in an order not even bound to the chronological order of the statements addition to the underlying triple stores. Any client relying on a particular order or uniqueness of the delivered results should take the care to achieve them himself. While such a functionality is easily implementable right inside the subscription service, we believe it will not be needed in most of the use cases we can imagine which led to the decision to leave it out in favor of implementation efficiency.

Leaving out the FILTER support at the current stage implies several limitations of the notification service. SPARQL achieves negative constraints only by means of FILTER(!BOUND(?)) filters. Thus, by disabling the FILTER support we turn notifications into strictly monotonous system. In other words, a statement that was once reported to meet the

requirements of a subscription will continue to meet them forever (or until it is removed from the triple store where it resides). In contrast, a non-monotonous notification system would be a lot more complicated due to the need to report not only statements that compounded the solution of a query but also the statements that stopped compounding any. In terms of implementation efficiency such a non-monotonous behavior would require a lot more computational resources and is of dubious value which brought the choice of a monotonous system and therefore the disabling the FILTER capabilities of graph patterns.

The purpose of the notification services is to enable the efficient and timely discovery of newly added RDF data. Therefore it should be treated as a mechanism for giving the client a clue whenever certain new data is available not as an asynchronous SPARQL evaluation system. The basic single triple pattern matching functionality was extended into a more complicated graph pattern matching in order to enable constraints on the structure of the triple surrounding graph. Further SPARQL-based extensions will be considered in future versions of the notification services but are not supported at the moment.

Inside OWLIM, where basic notification services are implemented, a subscription graph pattern GP = TP1,TP2,...,TPN containing N triple patterns is broken into N graph patterns of the form:

- GP1 = TP1',TP2,...,TPN
- GP2 = TP1,TP2',...,TPN
- ...
- GPN = TP1,TP2,...,TPN'

where TPK' in GPK is the same triple pattern as TPK but is to be matched against newly added triples rather than against the whole triple store.

When a batch of new statements is committed to the triple store each statement from the new batch is matched against GPK (for K = 1,...,N) by first matching against TPK' = TPK and if match was detected also checking for solution of the remaining part of the graph pattern in the triple store. If the triple doesn't match GPK (for K = 1,...,N) the GPK graph pattern is not further evaluated. This provides an efficient strategy for reducing the solution space to only triples from the newly added batch. Apart from enabling the system to only notify about newly added statements this strategy is a lot more efficient than a standard SPARQL query evaluation against the whole triple store which enables real-time data discovery and subscriber notification.

A client of the notification services should be aware that in its initial implementation only subscriptions to graph patterns consisting of a single triple pattern are guaranteed to return complete results. This limitation is due to the distributed nature of the RDF data storage subsystem and is not trivial to overcome at the current initial stage of development. However, one should expect that singleton graph patterns will notify their subscriber for all the matching incoming triples.

At the level of the semantic space Web service (Figure 4), techniques and protocols defined in the scope of WS-Notifications will be applied to enable the use of notification services also in the case of remote access to the semantic space platform. WS-Notification is an OASIS standard and offers a family of specifications that define a standard Web services approach to notification using a topic-based publish/subscribe pattern [6]. It includes: standard message exchanges to be implemented by service providers that wish to participate in notifications, standard message exchanges for a notification broker service provider (allowing publication of messages from entities that are not themselves service providers), operational requirements expected of service providers and requestors that participate in notifications, and an XML model that describes topics.

4. Service Marketplaces

Service marketplaces in the form of data-driven process executions are a premium application scenario for semantic spaces. It requires coordination, shared access to data and graph pattern or SPARQL-based data preparation. Coordination is either done implicitly via blocking operations over shared data, via notification services based on graph pattern subscriptions (not yet implemented) or via a centrally controlling process execution unit, as proposed in this section. Although showcased here by means of the latter approach, ultimately notification services will automatically control the data flow and thus the execution of mash-up-style compositions.

Semantic Web services traditionally rely on ontology-based classifications for the typing of input and output concepts, and for the formalization of conditions and effects. Also the more recently emerging lightweight approaches to service descriptions such as SAWSDL still use conceptual links for the annotations of services. In particular, the descriptions of input and output messages are reduced to the linkage to a concept in an external ontology.

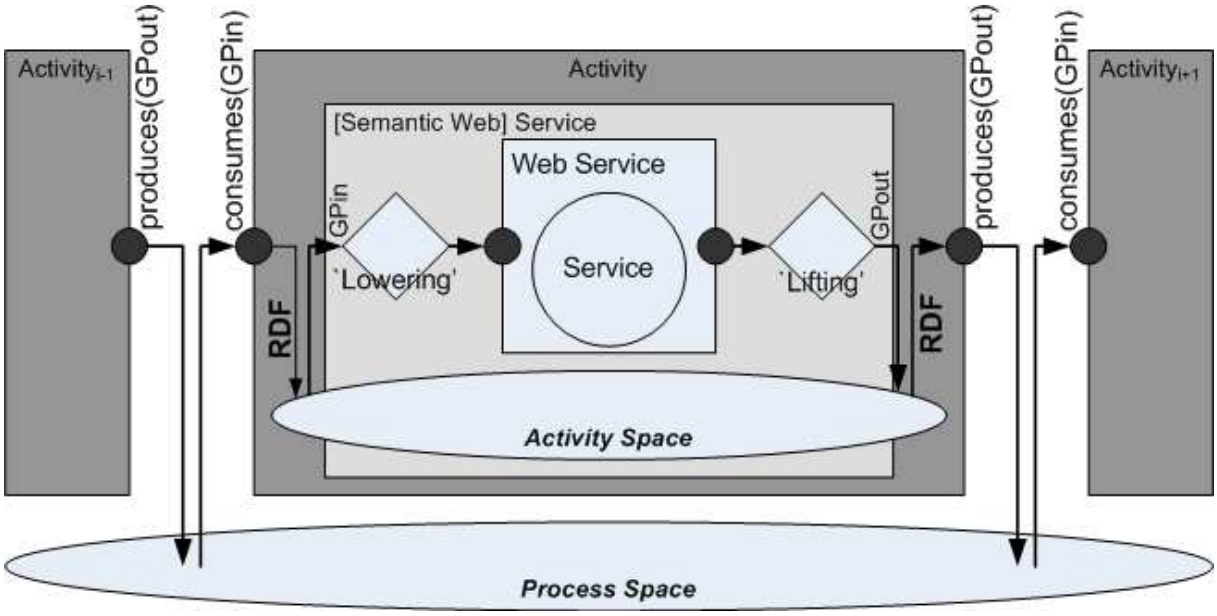

Figure 7: Semantic Web service composition in spaces

A core concept of the service marketplace approach is the fact that Semantic Web services now take RDF graphs as 'input message' and produce RDF as output too. This concept is schematically depicted on Figure 7. No matter what type of service is bound to the process, at the level of the Semantic Web service RDF is consumed and produced, and all communication is conducted at the semantic level. The description of the input and output, respectively, is then no longer given by linking some value to some concept in the ontology, but rather by a graph pattern which precisely describes the content of the expected input graph (condition), respectively the guaranteed output graph (effect). A graph pattern that describes the input to the GeoNames 'search' RESTful service is shown in Table 1,[1] as part of the semantic service description of the example process in Section 4.1. Pointing back to the goal of using notification services, the input graph pattern could directly be used as subscription pattern and thus serves two purposes: subscription and pre-condition.

Having the corresponding RDF data available in a process-specific semantic space (termed

[1] http://sws.geonames.org/search

'Process Space' on Figure 7) allows for the creation of SPARQL CONSTRUCT queries that fetch and prepare the required RDF statements. In the simplest case, the one depicted in Figure 7, all the required and produced RDF data is shared in a process-owned space against which the SPARQL queries are executed. Generalizing this setting, without altering any of the core concepts, the input data could also be derived by more complex CONSTRUCT queries over public data sources, or by means of Semantic Web Pipes [7] that aggregate data from various sources on the Web or the Linked Open Data cloud.[2] The RDF data prepared in that way is maintained in a so-called 'Activity Space' that stores the data specific to the execution of one Web service. Firstly, the 'Activity Space' contains exactly the data that is expected by a service, and secondly, it provides a simple means to create some state across the execution of an activity. The production process for GPout does not only consider the RDF data produced by the service execution, but can also take into account data that was prepared for the service invocation.

The use of semantic spaces as shared and persistent data management layer has thus several advantages. Processes, i.e. service mash-ups can be executed in time and reference decoupled manners. Subsequent services do not directly invoke each other, nor do they have to know of their precise existence. Execution is governed by coordinated access to the shared knowledge, and not by invocation. In tuplespaces systems [8], coordination is realized by means of blocking operations that only fire once a required piece of information is available in the shared space. Blackboard system rely on a coordinator agent that passes a token along to all participators [9]. In the scope of semantic spaces, publish-subscribe style notification services that exploit graph patterns as subscriptions are investigated as a means to trigger the invocation of a service [10]. As a direct consequence of this implicit invocation of services within a process, there is no need to ship data around between Semantic Web services. In fact, any service selects by construction precisely the data from the space that is required, and as such, the input to one service is a priori independent of the output of the predecessor. In other words, the execution of a process does not really require the specification of data flow.

An important missing piece to the execution of service compositions over semantic spaces relates to the fact that most Web services do still not accept or produce RDF data. Although more and more services rely on RDF directly, or REST principles for invocation, XML is still the predominant format for data transfer over the Internet. This is even the case for services that are internally manipulating RDF data, such as for example the GeoNames services that operate over Linked Data. In order to invoke a Web service, it is thus still necessary to transform from RDF to the expected data format of the service implementation, and to map the output of the service back into RDF. A Semantic Web service does consequently not only specify in its description what graph patterns it consumes and produces, respectively, but also how the input RDF is 'lowered' to the expected data format of the service, respectively how the output of the service is 'lifted' back to RDF. Again, if services consume RDF, 'lowering' and 'lifting' are obsolete and the input message – RDF data – is directly fed from the 'Activity Space' to the service. In all other cases the lowering box prepares the appropriate call. This can be done by help of a simple SELECT query that binds values to the query variables in a URL, or by more heavy-weight XSLT or XSPARQL transforms for XML-based services;[3] e.g., services expecting SOAP messages. The different types of lowering are illustrated in the example below.

[2] http://www.linkeddata.org

[3] http://www.w3.org Submission/2009/01/

4.1 Process Execution Example over Semantic Spaces

In order to evaluate our conceptual approach, we have carried out a proof-of-concept implementation. To illustrate the implementation we designed a hypothetical scenario that relies on the composition of three services: a real existing RESTful service that produces RDF data, a real one that returns an XML message and a locally implemented service that relies on SOAP messages. We chose not to use an existing service only to avoid real-life side effects. The example thus composes three activities that represent each of the three services.

The first service is from GeoNames. It takes the name of some geographic feature, and a return type as query variables: http://ws.geonames.org/search?type=rdf&name=Innsbruck. By putting the type to RDF, the service returns RDF/XML, rather than some proprietary XML document, that can be written directly back into the activity space without the need for lifting. One possible semantic description of this service is shown in Table 1. There are references given to the condition on the input of the service, the effect of the output, and a link to a lowering schema mapping; there is no need for a lifting schema, as the service natively returns RDF. In our SPARQL-minded approach, the condition and effect of a service are given by graph patterns, as they are defined by the SPARQL specification. In fact, the graph patterns that are part of the service description relate directly to the consumes and produces part, respectively, of the activity description – compare, for example, the value of the condition in Table 1 with the consumes statement of the first activity in Table 2. The process specification is needed by the engine that controls the execution of the process; more details are given in [11].

The lowering for the ‘search' service is very simple in the given case. All that is needed is the name of the location to ‘search’, which becomes part of the service URL as query string. Hence, the sole variable binding of the SPARQL SELECT query “SELECT ?name WHERE {?x geo:name ?name.}." that is specified in the lowering schema mapping file in the service description of Table 1 becomes the ?name variable of the service URL. This SELECT query is executed against the activity space that is populated according to the aforementioned CONSTRUCT query (Table 2).

Table 1: Lightweight service description for GeoNames' search service

```
@prefix sparql: <http://www.w3.org/TR/rdf-sparql-query/#> .
@prefix wsl: <http://www.wsmo.org/wsmo-lite#> .
@prefix geo: <http://ws.geonames.org/> .
@prefix sawsdl: <http://www.w3.org/ns/sawsdl#> .

geo:search rdf:type wsl:Service;
   sawsdl:modelReference _:cond, _:effect;
   sawsdl:loweringSchemaMapping
            <http://localhost:8080/search.sparql> .
_:cond rdf:type wsl:Condition;
   rdf:value "?x geo:name ?name ."^^sparql:GraphPattern .
_:effect rdf:type wsl:Effect;
   rdf:value "?x wgs84:lat ?lat;
          wgs84:long ?lng ."^^sparql:GraphPattern .
```

The second service is GeoNames’ ‘findNearByWeatherXML' service.[4] It takes latitude and longitude coordinates as input and returns the observations of the nearest weather station as an XML document, including temperature, humidity, clouds, atmospheric pressure, and wind direction and speed. The service description is similar to the one presented in Table 1, and

[4] http://ws.geonames.org/findNearByWeatherXML

we spare it here. The same accounts for the lifting and lowering, which is similar but simpler than the more interesting one presented in the context of the SOAP-based service (Table 4).

Table 2: Process specification

```
@prefix act: <http://www.example.org/SWServices/Activities#> .
@prefix proc: <http://www.example.org/SWServices/Process#> .
@prefix geo: <http://ws.geonames.org/> .
@prefix sparql: <http://www.w3.org/TR/rdf-sparql-query/#> .

_:p1 rdf:type proc:Sequence;
       proc:hasFirst _:a1;
       proc:hasNext _:x;
       proc:consumes "CONSTRUCT ..."^^sparql:construct .
_:a1 act:performs geo:search;
       act:consumes "PREFIX geo: <http://www.geonames.org/ontology#>
                          CONSTRUCT {?x geo:name ?name .}
                          WHERE {?x geo:name ?name .}"^^sparql:construct;
       act:performs "PREFIX wgs84: <http://www.w3.org/2003/01/geo/wgs84_pos#>
                         CONSTRUCT {?x wgs84:lat ?lat; wgs84:long ?lng .}
                          WHERE  {?x wgs84:lat ?lat; wgs84:long ?lng .}"^^sparql:construct .
_:x proc:hasFirst _:a2;
    proc:hasNext _:ch .
_:a2 act:performs geo:findNearByWeatherXML;
       act:consumes "CONSTRUCT ..."^^sparql:construct;
       act:performs "CONSTRUCT ..."^^sparql:construct .
_:ch proc:hasBranch _:cbr, _:br .
_:cbr proc:hasCondition "ASK ..."^^sparql:ask;
        proc:chosesFor _:p2;
        proc:asserts "CONSTRUCT ..."^^sparql:construct .
_:br proc:chosesFor _:p2;
     proc:asserts "CONSTRUCT ..."^^sparql:construct .
_:p2 proc:hasFirst _:a3 .
_:a3 act:performs <http://localhost:8080/axis/YP.jws>;
       act:consumes "CONSTRUCT ..."^^sparql:construct;
       act:performs "CONSTRUCT ..."^^sparql:construct .
```

The third service that is invoked depends on the weather information returned by GeoNames' weather service. For this reason, the process contains a 'choice' construct over the knowledge in the process space that was asserted by the preceding activities. In Table 2 the choice is represented by the blank node identifier _:ch and has two branches, a conditional one (_:cbr) and a default one (_:br). The conditional branch of the choice is implemented by means of the ASK query that is given in Table 3; it uses a temperature threshold to evaluate the condition.[5] Consequently, either the SOAP-based yellow pages service is invoked to query the phone number of a local restaurant, or under better weather conditions the contacts of some tennis club in the chosen city. As both branches of the choice trigger the same activity that wraps the service 'http://localhost:8080/axis/YP.jws', the difference in execution is given by the asserted triples of the branches. As stated above, the conditioned branch causes the request for a restaurant's contact information, and hence the type of business to look for that has to be communicated to the yellow pages service is "Restaurant", otherwise "Tennis".

[5] The predicate 'temperature' matches the predicate of the lifted output of the weather service.

Table 3: The 'choice' construct: ASK condition and CONSTRUCT-based assertion

```
PREFIX wth: <http://www.example.org/weather#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

ASK { ?x wth:temperature ?t .
      FILTER (?t < "10"^^xsd:integer) .}
```

```
PREFIX yp: <http://www.example.org/yellowpages#>

CONSTRUCT {_:yp1 yp:loc "Innsbruck";
                 yp:type "Restaurant" .}
WHERE {}
```

The triples which are asserted by the choice construct match the condition graph pattern of the yellow pages service: "?x yp:loc ?l; yp:type ?t .". The choice thus ensures that the input data, which is required by the last activity of our composition, is available in the process space, and can be moved into the activity space. From there the activity's consumes construct is invoked and the semantic data is lowered to the required SOAP message (Table 4), which is sent to the yellow pages service. The response XML is analogously lifted back to RDF and written back to the process space as final output of the process. In our example a graph is constructed that describes a restaurant or tennis club entry with a name, phone number and address.

Table 4: Lowering schema for the yellow pages service

```
<lowering>
  <sparqlQuery>
    PREFIX yp: <http://www.example.org/yellowpages#>
    SELECT ?loc ?type
    WHERE { ?x yp:loc ?loc; yp:type ?type .}
  </sparqlQuery>
  <xsl:transform version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:yp="http://www.example.org/yellowpages#">
    <xsl:output method="xml" version="1.0" ... />
    <xsl:template match="/sparql">
      <yp:TelephoneNumberRequest>
      <xsl:for-each select="results/result">
        <yp:loc>
        <xsl:value-of select="binding[@name='loc']/literal"/>
        </yp:loc>
        <yp:type>
        <xsl:value-of select="binding[@name='type']/literal"/>
        </yp:type>
      </xsl:for-each>
      </yp:TelephoneNumberRequest>
    </xsl:template>
  </xsl:transform>
</lowering>
```

This concludes the proof-of-concept example. Out of a set of lightweight semantic service descriptions with conditions and effects in form of graph patterns, and a formal specification of a process, we construct a composition that is entirely based on well-established (Semantic) Web technologies: RDF, SPARQL and XSLT/XSPARQL for lifting and lowering, if required. Moreover, the execution of the process is entirely governed by access to the

shared RDF data in the process space, an execution-specific semantic space. By coordinating the access to the semantic space, arbitrary marketplaces can be created out of loosely-coupled service mash-ups. Such data-minded processes are a premium application scenario for semantic spaces, and bring service coordination much closer to Web computing. Consequently, semantic spaces allow for Web2.0-style service marketplaces entirely based on semantic artefacts.

5. Conclusions

In this second specification round, we concentrated on two main aspects to optimize semantic spaces: i) bringing semantic spaces closer to standardized RDF processing and storage frameworks, and ii) enabling mash-up-style service executions on top of semantic spaces. The former is mainly done by aligning the semantic space interaction models with the RDF2Go framework. RDF2Go provides an abstraction over triple (and quad) stores, which allows developers to program against quasi-standardized RDF2Go interfaces. The latter is fundamentally driven by the idea that Semantic Web services should communicate semantics in the form of RDF. Activities within a process read and write RDF triples from a shared space. This allows workflows to be largely asynchronously executed, as there is no need anymore to call one activity from its predecessor. Triples can be consumed from the space and produced to the space at will, as long as the triple sets (graphs) match the input graph pattern and output graph patterns of a Semantic Web service. In the given example, the patterns are modelled as pre-conditions and effects respectively.

Besides these two main achievements, this deliverable present further improvements that will be implemented during the next period, up to month M30. The P2P overlay interfaces are also aligned with RDF2Go, which yields that advantage of having a pluggable approach both at the semantic space API level and at the storage level, where any RDF2Go-compliant repository can now be bound to the space implementation. In terms of distribution, the updated semantic spaces are also more flexible in terms of implementation. The infrastructure is now able to cope with single-repository spaces and distributed P2P-overlay-based spaces, and the implementation is further pushed towards high performance cluster realizations. Last but not least, work is ongoing in profiting form listeners and notification services that are installed at the level of the repositories in order to expose notification services through the semantic space API. Subscriptions to triple patterns are now registered at the storage layer where they are optimally processed, and results are tunnelled through the distributed semantic spaces infrastructure back to the user.

All in all, the updated semantic space infrastructure will be more standards-compliant and more flexible in terms of configuration possibilities. This evolution ensures a multi-purpose and highly horizontal infrastructure for the realization of distributed and dynamic marketplaces of semantic data.

© SOA4All consortium

References

[1] R. Krummenacher, I. Toma, Ch. Hamerling, J.-P. Lorre, F. Baude, V. Legrand, Ph. Merle, C. Ruz, C. Pedrinaci, D. Liu and T. Pariente Lobo: SOA4All Reference Architecture Specification, SOA4All Project Deliverable D1.4.1A, March 2009.

[2] R. Krummenacher, I. Filali, F. Huet and F. Baude: Distributed Semantic Spaces: A Scalable Approach To Coordination, SOA4All Project Deliverable D1.3.2A v1.1, August 2009.

[3] R. Krummenacher, M. Fried, F. Huet, I. Filali, L. Pellegrino, and Ch. Hamerling: Distributed Semantic Spaces: A First Implementation, SOA4All Project Deliverable D1.3.2B, August 2009.

[4] K.G. Clark, L. Feigenbaum and E. Torres: SPARQL Protocol for RDF, W3C Recommendation, January 2008.

[5] The Hadoop Project, http://hadoop.apache.org/

[6] St. Graham, D. Hull and B. Murray: Web Services Base Notification 1.3 (WS-BaseNotification), OASIS Standard, October 2006.

[7] D. Le-Phuoc, A. Polleres, Ch. Morbidoni, M. Hauswirth and G. Tummarello: Rapid Prototyping of Semantic Mash-Ups through Semantic Web Pipes, 18th World Wide Web Conference, April 2009: 581- 590.

[8] D. Gelernter: Generative Communication in Linda, ACM Transactions on Programming Languages and Systems 7(1), January 1985: 80 -112.

[9] R. Engelmore: Blackboard Systems, Addison-Wesley Publishers, November 1988.

[10] R. Krummenacher, B. Norton, E. Simperl and C. Pedrinaci: SOA4All: Enabling Web-scale Service Economies, 3rd Int'l Conference on Semantic Computing, September 2009.

[11] R. Krummenacher, B. Norton and A. Marte: [Semantic Web] Services: Comprehension and Composition, Submitted to Extended Semantic Web Conference, 2010.