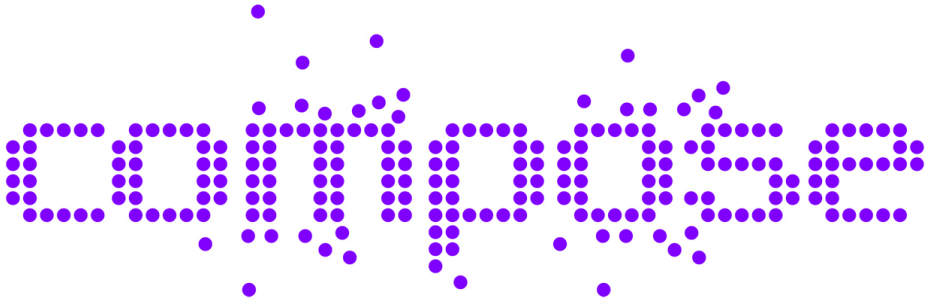

Collaborative Open Market to Place Objects at your Service

D3.1.1.1

Dynamic large-scale service discovery – First prototype

Project Acronym	COMPOSE	
Project Title	Collaborative Open Market to Place Objects at your Service	
Project Number	317862	
Work Package	WP3.1 Service Management	
Lead Beneficiary	IBM	
Editor	Yoav Tock	IBM
Reviewer	Rafael Giménez	BDIGITAL
Reviewer	Lukasz Radziwonowicz	FOKUS
Dissemination Level	PU	
Contractual Delivery Date	31/10/2013	
Actual Delivery Date	31/10/2013	
Version	V1.0	

Abstract

The service discovery infrastructure aims to provide a common registry that keeps track of all service objects and services hosted in the COMPOSE platform as well as other publicly available services and Web APIs. As part of the service management work package, it provides interfaces for updating and querying this registry, and specifically discovering services using diverse search criteria exploiting, among others, the semantic descriptions of services.

The service registry and discovery engine are based on semantic technology which supports the representation of services in a way such that automated reasoning can be directly applied in order to provide advanced discovery capabilities.

This document aims to accompany the initial prototype of the communication system rather than provide a detailed design document of the demonstrated system.

Document History

Version	Date	Comments
V1.0	31/10/2013	Submitted

Table of Contents

List of Figures

Acronyms

Acronym	Meaning
COMPOSE	Collaborative Open Market to Place Objects at your Service
RDF	Resource Description Framework
OWL	Web Ontology Language
TDB	(Jena) Triple Data Base
SKOS	Simple Knowledge Organisation System

1 Introduction

The service management work package is based on three main components (see Figure 1):

1. An advanced linked services discovery engine, whose job is to discover distributed and heterogeneous COMPOSE entities. The service discovery engine is layered on top of a service registry, which exploits information retrieval and semantic search and storage technologies. These two sub-components, emphasized in Figure 1, are *the focus of this deliverable*.
2. An advanced service recommender system, which is in charge of suggesting new relevant services based on users' previous interactions, similarity between services, and other non-functional properties such as performance, trust, etc.
3. An assisted service composition engine, which is meant to help users create new composite services by (semi) automatically combining existing services to obtain the desired functionality.

Both the service recommender and the service composition engines leverage the service discovery engine.

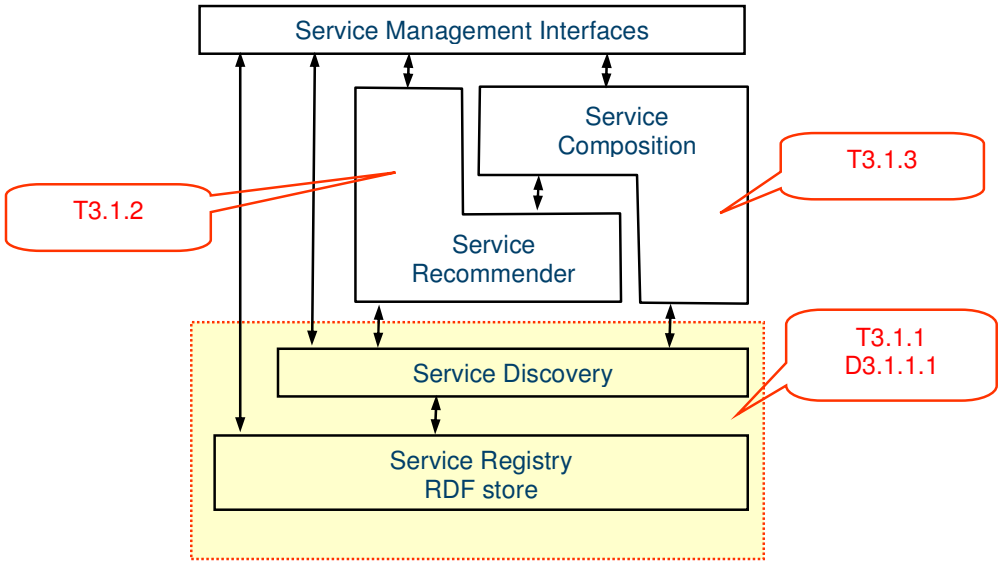

Figure 1 – A simplified logical view of service management components

The main function of the service discovery engine is to provide advanced, scalable, and efficient service discovery for end users and applications. Most challenging in this regard is the need to support third party applications, such as the composition engine from T3.1.3, to efficiently identify services that match certain criteria such as dataflow compatibility, a given categorisation of service's functionality, etc. The engine must therefore be able to evaluate hundreds of discovery requests over the registry with sub-second response times. To this end,

the discovery engine implements a number of search algorithms in order to rapidly assess the level of match with respect to a given request, rank the filtered results, and eventually return the discovery result. In its current version, the discovery algorithms are essentially based on logic operators over the semantic descriptions of services in order to compute the matching degrees. In future versions more relaxed matching operations may be introduced enabling as well the use of approximations to the matching of service requests.

The service discovery engine is layered on top of a service registry, which provides basic CRUD (Create, Read, Update, and Delete) operations of service descriptions. The backend used for storing these descriptions is implemented using an RDF store [1]. The RDF store and the upper layers are connected using the standard SPARQL query, update, and graph store protocols [2]. This design decouples the storage sub-component from the upper layers, which implement service management specific functionality.

With the current prototype we aim to demonstrate the main functional aspects which are necessary for the next step – the first integrated COMPOSE prototype (at month 18). The functionality needed for this aim is the basic ability to register, remove, and modify the attributes of a given service, as well as the ability to discover other services deployed in the platform. We therefore concentrate on two aspects: interface design, and correct semantics. Scalability, fault tolerance, and performance issues are deferred to the second delivery according to the plan (D3.1.1.2). In terms of interfaces, we focus on the CRUD (create, read, update, delete) API of service management, and the discovery API.

The main purpose of this document is to accompany the technical demonstration and provide the necessary background information to be able to put it into the right perspective within the COMPOSE platform. It is not intended to be a full-fledged design document of this component.

2 Prototype overview - short description

In order to support the efficient search and discovery of services in COMPOSE, we have adopted and extended iServe [3], an open platform for publishing semantic descriptions of services originating from the EU project SOA4All. The service registry is an Open Source project under Apache 2 license continuing the earlier work on SOA4All. The code, issue management system are publicly available at https://github.com/kmi/iserve. Additionally a dedicated Web site providing technical documentation of the latest version is available at http://kmi.github.io/iserve/latest/. Further details on how to obtain and compile the system are available therein as well as, for completeness, in Appendix A.

Despite being based on an already functional service registry, this prototype has incurred substantial development in order to adapt and extend the software to fulfill the requirements for COMPOSE. A considerable part of these extensions and improvements play a fundamental role in the scalability, extensibility and future usability and exploitation of the software. Appendix B provides a more detailed account on some of the main changes that have been carried out.

The essence of the approach followed by iServe is the use of import mechanisms for a wide range of existing service description formalisms (e.g., WSDL, WSMO-Lite, OWL-S, etc) to automatically transform and expose service descriptions as Linked Data [10]. Once transformed, the resulting service descriptions, which we refer to as Linked Services [11], are expressed in terms of a simple RDFS model, Minimal Service Model (MSM), which captures the essence of services and enables attaching semantic annotations to them. The reader is referred to D1.3.1 for more details on the conceptual model used.

This first prototype provides complete functionality, albeit not yet optimized, for most essential features. In particular, the registry provides CRUD operations over service descriptions, ontologies, and additional related documents. Ontology management is automatically controlled by an intelligent crawler that identifies the ontologies that need to be obtained and interpreted (or deleted) to support service discovery appropriately.

On top of this lower management layer, the registry implements service discovery algorithms supporting the identification of services based on ontology reasoning over service interfaces (i.e., inputs and outputs) and functional categorizations of services using Simple Knowledge Organization System (SKOS) schemes or taxonomies [12]. The engine supports additionally the definition of combinations of discovery filters, enabling thus applications to request services that match all filters (Intersection), some filters (Union), or even the mutual exclusion of some filters (Subtraction).

In order to better enable the extension and use of the software, much care and effort has been devoted to creating modular components and a range of APIs targeting the different uses envisaged. For externally provided functionality the registry offers both a simple yet versatile Java API enabling embedded integration of third –party software tightly bound to the registry, a RESTful interface enabling more decoupled integration over the Web, as well as an embedded Web graphical interface providing direct access to the registry to users. Out of these different interfaces which are covered in more detail in later sections of this deliverable, the Java API is of outmost importance for the integration of components from COMPOSE, notably the composition engine which has driven to a large extent the establishment of target performance metrics.

2.1 Architectural responsibilities and interactions

Two main roles are responsible for interacting with the service management components: the "developer" and the "platform provider" (see

Figure 2).

A "developer" is a COMPOSE user which develops new applications and services, pushes them into the platform, and then manages their lifecycle (i.e. start, stop, remove, etc). These actions, performed by a developer on his own applications and services, will result in updates to the service registry. The interactions that cause a service datum to be updated in the registry are mediated by the COMPOSE controller, which exposes a REST API and processes all the calls from the developer toolset (command-line, GUI, IDE, etc.).

A service developer may develop composite services, and thus bind to other third party services. In order to find those and decide which to use, he will use the discovery, recommendation, and composition engines. These engines allow read only access to the service registry and are directly exposed to the developer toolset.

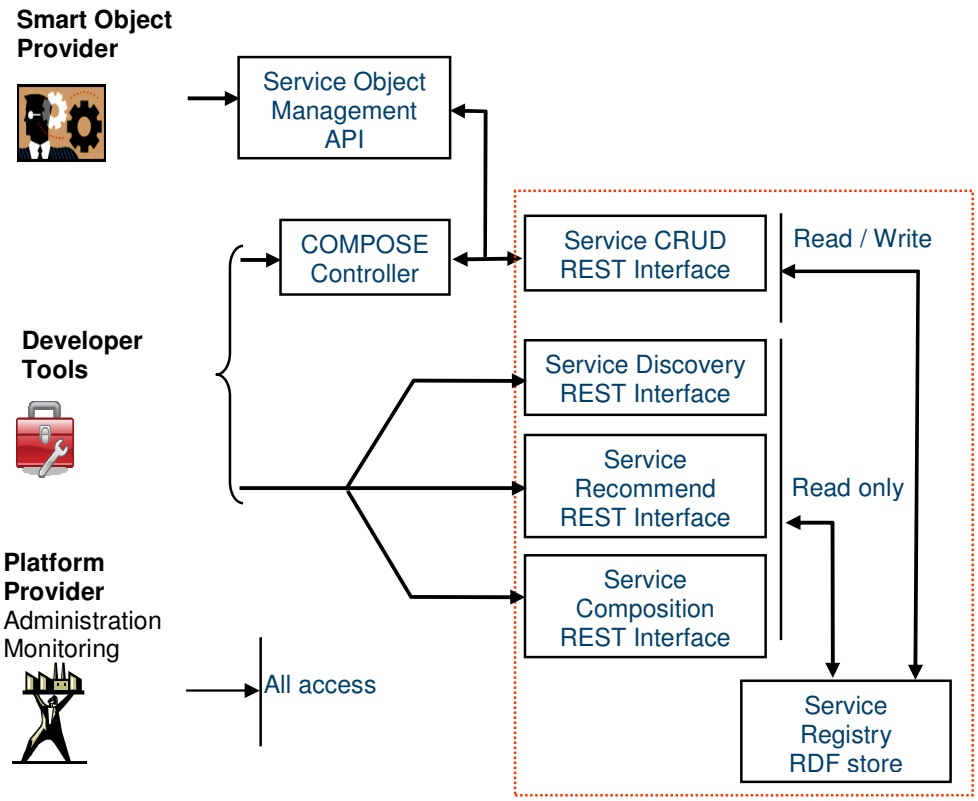

Figure 2 – Architectural interactions of COMPOSE components with service management interfaces

The action of actually binding a service to another service is done by the COMPOSE controller, and again is accompanied by updates to the service registry.

The platform provider has full administrative access to all the interfaces and internal components. The platform provider also collects statistics and monitors the activity of the service management components.

2.2 Lower part (RDF store) - high level design

The RDF store is the heart of the service registry. It stores service descriptions as RDF graph data. Essentially, each service is modelled as a named graph, which contains all the semantic information about its function and properties. This data forms the master record for services hosted by the COMPOSE platform.

Above the storage layer there are layers that implement SPARQL query and update, and a data access layer that enables the data to be queried and managed using the SPARQL suite of protocols.

There are many products and open source projects which implement RDF manipulation libraries, the SPARQL protocol, and RDF store. We chose to base most of our solution upon the Apache Jena [4] framework, because of its license (Apache 2), governance, popularity, robustness, and vibrant community. We extend and enhance the Jena TDB storage backend, and make it into a replicated, highly available RDF store. We chose the replication technology to suit the specific needs of the COMPOSE platform.

2.2.1 Usage analysis & requirements

Data store volume

We estimate that each service will be represented by a graph containing an order of 50 triples, with each triple taking an order of 1KB. A single service can thus be represented by 50KB (this is a generous estimate). One million services would consume an order of 50GB of data. This volume is relatively modest, and can easily be accommodated within the disk of a single server.

Data access pattern

Services are first written when they are pushed into the COMPOSE platform. Subsequently, they are updated when there is a change in their state following life-cycle events like deploying, starting, stopping and deleting the service. Services are read following discovery, recommendation, and composition requests.

We estimate that a typical workload would be heavily dominated by reads. Certainly the number of read queries is larger then writes, since a service is "found" (read) far more times during its lifecycle than it is written. Moreover, discovery queries generally have to read much more data in order to produce a single useful result for a user, whereas changes to a service datum tend to be more focused.

We assume that the bottleneck to scalability will be the request rate, especially reads and long running queries, rather than data size.

Data persistence, fault tolerance, and high availability

Since the service registry is the master record for services deployed in COMPOSE, it should be made very robust. This means storing the data on persistent storage and replicating it several times. The registry should remain available and endure the failure of some limited number of replicas without relapse in service. The registry should endure a complete shutdown and restart of the cluster without loss of data.

Data consistency

The data store API should support strong consistency between read and writes. This is required for the service management CRUD API, which is keeping track of services in the platform. It should support transactional behaviour, and support a strong form of isolation level (equivalent to the SQL "serializable" isolation level). The discovery, recommendation and composition engines, which only read data, may work under less demanding requirements, for example snapshot reads, or eventually consistent reads.

2.2.2 Main functionalities introduced

The requirements introduced above lead us to a design which is based on a replicated triple-store. Every write is replicated to all the servers, using a Paxos based algorithm, described below.

Paxos replication

The Paxos replication algorithm is a quorum based algorithm where a write succeeds once a majority of the replicas accept it [5]. This style of replication can withstand F failures Given 2F+1 replicas. The advantage is that it can tolerate slow replicas, and can gain progress even if some replicas are engaged in long running queries. With careful design, read performance can scale linearly with the number of replicas.

Figure 3 shows a replicated state machine architecture with front ends (FE) which propose commands (e.g. writes) at any order. The Paxos layer decides on a global order of commands that have been agreed upon by a quorum of the replicas. The replica back-ends (BE) see an identical stream of commands. If the back-end is a database, the write command stream leave the database in an identical state in all replicas.

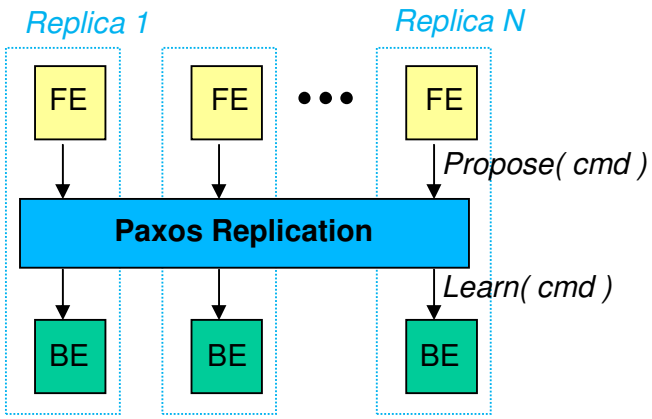

Figure 3 – Essentials of Paxos type replication.

During network partitions, only the partition containing a majority of the replicas will continue to accept commands (writes). Thus no conflict resolution is needed after a partition is fixed (at the cost of unavailability of writes in the minority partition).

This replication style fits the case where there is need for highly consistent replication, with support for scalable read performance, and support for long queries on some of the replicas.

The Paxos variant that we use [6] supports 'reconfiguration' - the set of replicas can be dynamically changed without stopping the cluster - a tricky issue that is often glossed over in competing approaches and implementations (e.g. [7]). This feature facilitates dynamic cloud deployments and geospatially dispersed deployments. Our implementation also provides a framework for snapshot management and automatic state transfer after failure and network partition.

Architecture of the service registry

The service registry architecture is composed of the Jena stack, with a custom replicated storage backend that we call "PaxDB". As Figure 4 illustrates, the top level is made of the Fuseki server, which enables HTTP access and implements the standard SPARQL protocols. Below Fuseki is the full set of Jena APIs. PaxDB essentially replicates TDB using a Paxos based replication approach. The PaxDB front-end captures all the write API calls and replicates them

across the cluster. The writes which have been accepted by a majority are then executed by the back-end against TDB, at exactly the same order in all the replicas. Reads can either be executed locally, which results in snapshot consistency (a legal but possibly outdated snapshot of the database); or after a synchronization with the cluster which results in "serializable" consistency (the strongest level of consistency).

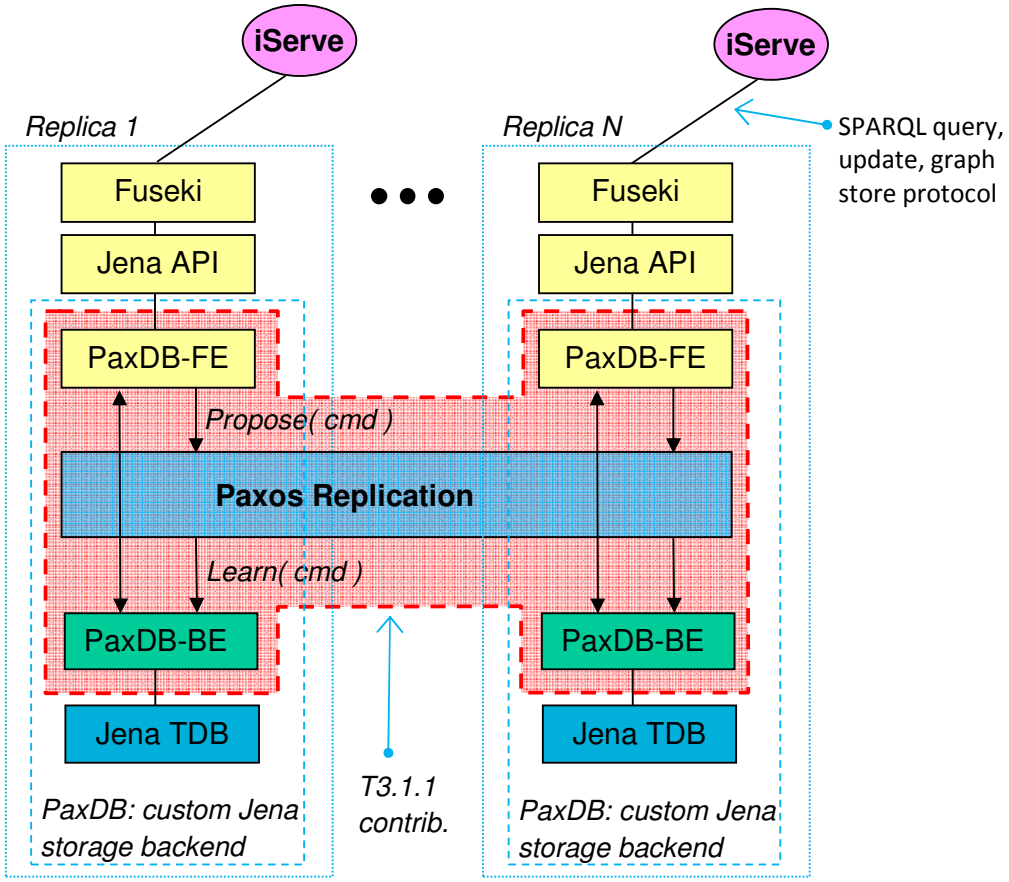

Figure 4 – Anatomy of the replicated service registry

2.2.3 What is in the first prototype

The focus of the first prototype is on functional aspects that would allow integration with the rest of COMPOSE. For the service registry this means producing a functional prototype that provides all the functional aspects but does not necessarily address the more advanced scalability and fault tolerance aspects. We therefore chose to implement the first prototype using a single node Jena stack with Fuseki at the top, and TDB as the storage layer (see Figure 5).

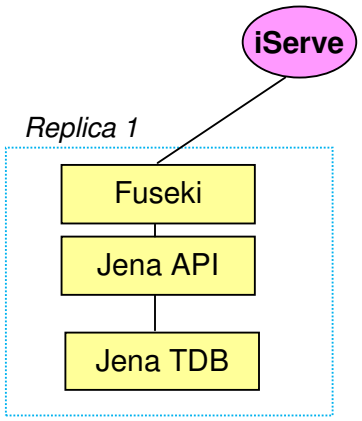

Figure 5 – Service registry first prototype

2.3 Upper part (service discovery) - high level design

The bottom layer, described in detail earlier on, is in charge of managing the registry's data that includes Service descriptions, related documents (e.g., original files), and the corresponding Ontologies. This layer essentially provides a RDF/S and OWL storage and reasoning support, document storage, as well as basic crawling facilities to e.g., automatically obtain referenced Ontologies. RDF/S and OWL storage and reasoning support is delegated to dedicated engines which are accessed at higher levels by means of the SPARQL 1.1 standard [2]. As a consequence, the reasoning capabilities of iServe and by extension the accuracy and refinement of the discovery functionality offered depend largely on the actual configuration of the underlying store.

File storage is currently supported directly by the file system although the actual implementation to be used could be switched easily in anticipation to more scalable solutions that could for instance delegate this to more scalable solutions, in the spirit of the Hadoop Distributed File System or Amazon S3, based on the runtime infrastructure.

On top of the data layer, iServe provides a storage and management layer that is in charge of:

1. supporting advanced and efficient access to the data layer,
2. supporting the import of service annotations in a variety of formalisms,
3. pre-processing services and documents for the indexing of services.

Finally, the third layer is in charge of providing advanced discovery and analysis functionality exploiting the data held by the registry.

Currently the code base includes the functionality for advanced discovery as a set of plugins. The Crawler and Analysis Engine modules are still work in progress and will be added in subsequent versions to populate the service registry with live data automatically crawled from the Web.

The functionality of the upper two layers is exposed as RESTful services so that both Web interfaces and external applications can easily use them. In the case of Service descriptions this therefore represents a Read/Write Linked Data API.

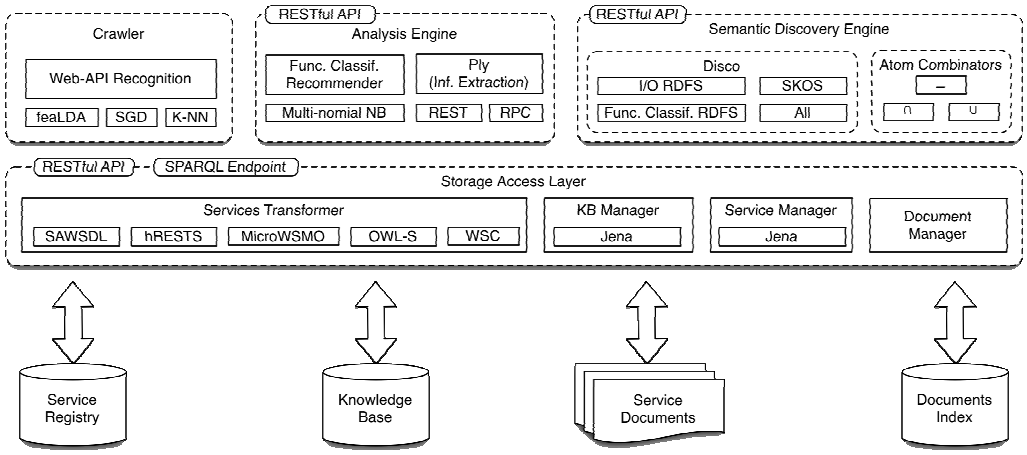

Figure 6 – Service discovery detailed view.

2.3.1 Main functionalities introduced

iServe provides two main means for its integration into applications, namely a direct Java API, and a RESTful interface. In the remainder of this section we briefly go into both interfaces providing simple examples of how 3rd party developers can make use of them. For further reference the reader is referred to the documentation of the software available at http://kmi.github.io/iserve/latest/.

Java API

Currently access to iServe's functionality is provided through two main entry points depending on whether you want to interact with its CRUD interface to add services, documents, or ontologies, or with its matching interface if you want to discover services.

CRUD Interface

Access to the CRUD interface is obtained by means of iServeFacade. Therein you will get access to the main data managers in charge of services, documents, and ontologies. Each of these managers provides the typical methods for adding, removing, or retrieving the entities they manage. For convenience iServeFacade provides methods that take care of automatically transforming services prior to importing them, keeping all data managers synchronized, and when necessary, obtaining remote ontologies referred to by the imported services.

Obtaining the Facade

```
iServeFacade facade = iServeFacade.getInstance();
```

Importing Services

```
InputStream in;
List<URI> servicesUris;
for (File ttlFile : msmTtlTcFiles) {
    in = new FileInputStream(ttlFile);
    servicesUris = facade.importServices(in, MediaType.TEXT_TURTLE.getMediaType());
}
```

Listing the Registered Services

```
List<URI> services = facade.getServiceManager().listServices();
```

Obtaining the Java Representation of a Service

```
URI svcUri = new URI("http://iserve.installation.org/services/id/1");
```

The Matching Interface

iServe provides a Matching API that can be used programmatically from Java. Although work is ongoing to provide a more advanced and richer API, you can already work with it within your applications. Should you wish to obtain a Matcher, you can simply do so by means of the uk.ac.open.kmi.iserve.discovery.api.impl.MatchersFactory. This factory takes care of locating Matcher implementations by means of a plugin mechanism, and will automatically create the plugin of your choice at runtime. Currently, the code provides a ConceptMatcher allowing you to find matches between concepts loaded in the server by relying directly on SPARQL queries. Further matchers are under development and will be released incrementally.

Below you have an example on how you can create a Matcher.

```
ConceptMatcher matcher =
MatchersFactory.createConceptMatcher(SparqlLogicConceptMatcher.class.getName());
```

Notifications

In order to enable applications and extensions to react upon changes to the registry, iServe implements a simple notification mechanism based on Guava's EventBus. Applications can register for notifications simply by doing:

```
facade.registerAsObserver(this);
```

This will ensure than any change on the Services, Documents or Knowledge Base Managers get automatically reported. Processing the events is simply ensured by implementing a method that takes the event we are interested in as sole argument. The method should be annotated with @Subscribe. See package uk.ac.open.kmi.iserve.sal.events for the list of events supported.

An example on how such a subscription method should be implemented is listed next.

```
@Subscribe
public void handleOntologyCreatedEvent(OntologyCreatedEvent event) {
    // Code handling the event...
}
```

3 External API - for use by 3rd parties

In this section we provide an overview of the REST APIs that are used by the other components in COMPOSE.

As introduced earlier, the registry follows the Linked Data principles for publishing service descriptions. Each element of the registry will therefore have its own de-referenceable URI that can be directly accessed via HTTP in order to obtain both human-oriented representations (e.g., HTML page) and machine-oriented representations (e.g., RDF, JSON).

The URI of elements stored in the registry are structured based on the following URI scheme:

- Graph Identifying the entire service description
  - **http://myServer.com/id/services/{svcId}**
- Service URI
  - **http://myServer.com/id/services/{svcId}/{svcName}**
- Operation URI
  - **http://myServer.com/id/services/{svcId}/{svcName}/{opName}**
- Message Content URI
  - **http://myServer.com/id/services/{svcId}/{svcName}/{opName}/{messageName}**
- Message Part URI
  - **http://myServer.com/id/services/{svcId}/{svcName}/{opName}/{messageName}/{messagePart}**

3.1 Registry Management Operations

Over all the URIs above the server supports GET operations in order to obtain the service details:

GET **http://myServer.com/id/services/{svcId}/{svcName}**

Currently, the server implements Content Negotiation in order for clients to select their preferred serialization format out of: RDF/XML, Turtle, XML, JSON, HTML.

Additionally as a particular extension of iServe to traditional Linked Data provisioning, the system supports also Write operations allowing users to modify (i.e., upload or delete) service descriptions. This type of functionality, currently under discussion for standardization within the W3C Linked Data Platform Working Group, is implemented in iServe in the same spirit, following strictly the REST principles.

Notably, the creation of a new service is achieved by doing:

POST **http://myServer.com/id/services/**	The service description should be submitted as a multipart form

Likewise, services can be deleted as follows:

DELETE **http://myServer.com/id/services/{svcId}/{svcName}**

3.2 Human-oriented Registry Browsing API

We provide additionally a simple and configurable browsing API that automatically generates queries to filter data from the registry and present it to the user through an HTML GUI. Among the methods currently defined we have:

- List Services
  GET **http://myServe.com/doc/services**
- Obtain Service Details
  GET **http://myServe.com/doc/services/{svcId}/{serviceName}**}
- List Operations
  GET **http://myServe.com/doc/operations**
- Obtain Operation Details
  GET **http://myServe.com/doc/services/{svcId}/{serviceName}/{opName}**}
- Obtain Messages for Operation
  GET **http://myServe.com/doc/services/{svcId}/{serviceName}/{opName}/messages**
- Obtain Message Details
  GET
  http://myServe.com/doc/services/{svcId}/{serviceName}/{opName}/{messageName
  }

3.3 Advanced Discovery API

The current high-level discovery API exposed through a RESTful interface is structured around 3 mains aspects. On the one hand the engine offers support for discovering services based on coarse-grained classifications of the functionality of services (e.g., weather forecast, traffic sensing, etc). On the other hand, the engine provides support for searching services based on their interface (i.e., the data consumed and produced). Finally, the system provides support for combining diverse discovery queries to create complex ones.

The results provided by each of these plugins are currently provided as Atom lists which is a widely supported standard on the Web.

3.3.1 Functional Classification Based Discovery

*GET **http://myServe.com/disco/{type}/func-rdfs?class={c}***

***Type** – parameter indicating the type of item to discover. The only values accepted are "op" for discovering operations, and "svc" for discovering services.**C** – multivalued parameter*

indicating the functional classifications to match. The class should be the URL of the concept to match. This URL should be URL encoded.

For example:

GET http://myServe.com/disco/{type}/func-rdfs?class=http://someServer/onto%2523TemperatureService

3.3.2 Input/Output Based Discovery

GET http://myServe.com/disco/{type}/io-rdfs?f={f}&i={i}&o={o}

***F –** type of matching. The value should be either "and" or "or". The result should be the set-based conjunction or disjunction depending on the value selected between the services matching the inputs and those matching the outputs.*

***I –** multivalued parameter indicating the classes that the input of the service should match to. The classes are indicated with the URL of the concept to match. This URL should be URL encoded.*

For example:

i=http://someServer/onto%2523Concept2

***O –** multivalued parameter indicating the classes that the output of the service should match to. The classes are indicated with the URL of the concept to match. This URL should be URL encoded.*

For example:

o=http://someServer/onto%2523Concept2

Complete Example:

GET http://myServe.com/disco/{type}/io-rdfs?f=and&i=http://someServer/onto%2523Concept2&o=http://someServer/onto%2523Concept2

3.3.3 Combinations of Discovery Queries

UNION

Union discovery queries provide the results that match any of the inner queries.

GET http://myServe.com/atom/union?f={f}

***F –** multivalued parameter indicating the inner discovery queries to use for performing the union. Each inner query can be specified with the path relative to the server as specified above. Note that URLs need to be URL encoded as well so in this case they are doubly URL encoded (one for the atom combination request, and another one for the actual discovery request).*

For example:

f=/data/disco/func-rdfs?class=http://someServer/onto%2523Concept1

&f=/data/disco/io-rdfs?o=http://someServer/onto%2523Concept2

INTERSECTION

Intersection discovery queries provide the results that match all the inner queries.

*GET **http://myServe.com/atom/intersection?f={f}***

***F –** multivalued parameter indicating the inner discovery queries to use for performing the union. Each inner query can be specified with the path relative to the server as specified above. Note that URLs need to be URL encoded as well so in this case they are doubly URL encoded (one for the atom combination request, and another one for the actual discovery request).*

For example:

f=/data/disco/func-rdfs?class=http://someServer/onto%2523Concept1

&f=/data/disco/io-rdfs?o=http://someServer/onto%2523Concept2

SUBTRACTION

Subtraction discovery queries provide the results matching an inner query minus those matching the other inner query.

*GET **http://myServe.com/atom/minus?f={f}***

***F –** multivalued parameter indicating the inner discovery queries to use for performing the union. Each inner query can be specified with the path relative to the server as specified above. Note that URLs need to be URL encoded as well so in this case they are doubly URL encoded (one for the atom combination request, and another one for the actual discovery request).*

For example:

f=/data/disco/func-rdfs?class=http://someServer/onto%2523Concept1

&f=/data/disco/io-rdfs?o=http://someServer/onto%2523Concept2

4 Initial performance testing

For this initial prototype we focused on verifying correctness and checking the interfaces. iServe contains a comprehensive test suite (see Appendix A). Initial performance numbers are taken from a development setup: a VM on Oracle VirtualBox, hosted on Windows 7 (1 cpu, 2GB memory). The VM contains both the iServe server, Apache server, and the Fuseki server. The Fuseki server is configured to work with TDB, using transactions, and commit every write to disk.

Note that these performance numbers are not indicative of a production deployment.

Test	Throughput [ops/sec]	Latency [ms]
Create service	0.47	2130
Read service	24	21.6

5 Future directions

In this section we consider some work items which are deferred beyond the current delivery.

5.1 Cloud deployment

The runtime system of COMPOSE is based on the Cloud Foundry (CF) Platform as a Service (PaaS) cloud (see [8]). Most internal components in COMPOSE need to be adapted to be hosted and/or consumed in a cloud environment. For any COMPOSE logical unit, that means decomposing into components which are either CF-services or CF-apps. Roughly speaking, elements which are accessible to the web, and are hosted on stateless VMs, are CF-apps. CF-apps store data only through platform provided data storage services (like MongoDB). CF-apps are typically web applications hosted on Java application servers, NodeJS servers, etc. Components which run on statefull VMs, store data, and expose a client API are generally CF-services. Salient examples for CF-services are MongoDB, MySQL, etc.

As for the service registry and discovery units, the REST APIs for CRUD and discovery need to be hosted as a CF-app, because they need to be accessible from the web. The service registry (RDF store) is most logically hosted as a CF-service. Some aspects of the iServe server can be hosted as a CF-app as long as the persistent state is stored through a platform service like MongoDB.

5.2 Highly scalable and efficient discovery

The main goal of the second delivery D3.1.1.2 in month 30 is to handle the scalability, fault tolerance, and performance. Scalability and fault tolerance will essentially be handled by the lower layer that implements the RDF store, as shown in Figure 4. Performance improvements will on the other hand require work at both levels covering both fast SPARQL query response times, as well as high-level indexing in order to minimise as much as possible the time for evaluation of queries.

5.3 OSLC

OSLC – Open services for lifecycle collaboration [9], is an emerging community effort to provide specifications that will facilitate software to work with other software. It is intended to

be used in many domains, from architecture management, product line management, configuration management and automation. The OSLC specification is based on RDF and linked data, and is compliant in spirit with our approach. Our plan is to explore the links to the OSLC specification and emerging standards, and check whether we can adopt relevant parts of this spec and possibly contribute to it.

6 References

[1] W3C RDF definition. http://www.w3.org/RDF/

[2] SPARQL 1.1 Overview. W3C Recommendation 21 March 2013. URL: http://www.w3.org/TR/sparql11-overview/

[3] Pedrinaci, C., Liu, D., Maleshkova, M., Lambert, D., Kopecky, J. and Domingue, J. (2010) **iServe: a Linked Services Publishing Platform**, Workshop: Ontology Repositories and Editors for the Semantic Web at 7th Extended Semantic Web Conference.

[4] Apache Jena. A free and open source Java framework for building Semantic Web and Linked Data applications. URL: http://jena.apache.org

[5] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.

[6] BORTNIKOV, V., CHOCKLER, G., PERELMAN, D., ROYTMAN, A., SHACHOR, S., AND SHNAYDERMAN, I. Frappe : Fast replication platform for elastic services. In *ACM LADIS* (2011).

[7] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *PODC* (2007), pp. 398–407.

[8] Cloud Foundry – Open PaaS. http://www.cloudfoundry.com/

[9] OSLC: Open Services for Lifecycle Collaboration. URL: http://open-services.net/

[10] C. Bizer, T. Heath, and T. Berners-Lee, "Linked Data - The Story So Far," *International Journal on Semantic Web and Information Systems (IJSWIS)*, 2009.

[11] Pedrinaci, C. and Domingue, J. (2010) **Toward the Next Wave of Services: Linked Services for the Web of Data**, Journal of Universal Computer Science 16(3).

[12] S. Bechhofer and A. Miles, Eds., "SKOS Simple Knowledge Organization System Reference," *W3C Recommendation*, 18/08/2009.

Appendix A – Building the system

This appendix describes the steps taken in order to build a working iServe system from the publically available iServe distribution. iServe requires a SPARQL endpoint, and in this appendix we use the Apache Fuseki SPARQL endpoint.

Assumptions

- In the appendix we assume you are running under Linux, however the steps should be very similar for other platforms.
- Other packages we assume are installed on your system:
  - git
  - maven
    - It is recommended to increase the memory of the maven JVM:
      export MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=256m"
      and restart your shell.
  - Sun JDK 1.6 is installed on the system

1st step: Fuseki

- Download
  - Fuseki can be downloaded from http://archive.apache.org/dist/jena/binaries/. The latest version we have used is http://archive.apache.org/dist/jena/binaries/jena-fuseki-1.0.0-distribution.zip
- Install
  - Extract the downloaded archive, e.g. if it as a .zip file use 'unzip <filename>'
  - This will create a directory with a name similar to 'jena-fuseki-1.0.0'
- Configure - Create a Fuseki config file
  - Within the top-level Fuseki directory, create file named fuseki4iserve.ttl with the following contents:

```
@prefix :       <#> .
@prefix fuseki:  <http://jena.apache.org/fuseki#> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
@prefix tdb:    <http://jena.hpl.hp.com/2008/tdb#> .
@prefix ja:     <http://jena.hpl.hp.com/2005/11/Assembler#> .

[] rdf:type fuseki:Server ;
  # Timeout - server-wide default: milliseconds.
  # Format 1: "1000" -- 1 second timeout
  # Format 2: "10000,60000" -- 10s timeout to first result, then 60s timeout to for rest of query.
  # See java doc for ARQ.queryTimeout
  # ja:context [ ja:cxtName "arq:queryTimeout" ;  ja:cxtValue "10000" ] ;
  # ja:loadClass "your.code.Class" ;

  fuseki:services (
    <#service_tdb_all>
  ) .

# TDB
[] ja:loadClass "com.hp.hpl.jena.tdb.TDB" .
tdb:DatasetTDB  rdfs:subClassOf  ja:RDFDataset .
```

```
tdb:GraphTDB                                    rdfs:subClassOf                 ja:Model        .

## Updatable TDB dataset with all services enabled.

<#service_tdb_all> rdf:type fuseki:Service ;
   rdfs:label                "TDB Service (RW)" ;
   fuseki:name                 "ds" ;
   fuseki:serviceQuery          "query" ;
   fuseki:serviceQuery          "sparql" ;
   fuseki:serviceUpdate         "update" ;
   fuseki:serviceUpload         "upload" ;
   fuseki:serviceReadWriteGraphStore     "data" ;
   # A separate read-only graph store endpoint:
   fuseki:serviceReadGraphStore      "get" ;
   fuseki:dataset         <#tdb_dataset_readwrite> ;
   .

<#tdb_dataset_readwrite> rdf:type      tdb:DatasetTDB ;
   tdb:location "--mem--";
   #    tdb:location "DB" ;
   tdb:unionDefaultGraph true ;
##     # Query timeout on this dataset (milliseconds)
##     ja:context [ ja:cxtName "arq:queryTimeout" ;  ja:cxtValue "1000" ] ;
##     # Default graph for query is the (read-only) union of all named graphs.

    .
```

- Run
  - To run Fuseki use the command line: ./fuseki-server --config=fuseki4iserve.ttl
  - You now have a SPARQL endpoint running.

2nd Step: iServe

- Download
  - Retrieve the latest iserve distribution using the command:
  - git clone https://github.com/kmi/iserve.git
  - The latest iserve distribution should now reside in the subdirectory 'iserve'
- Configure
  - Configure iserve by creating the file
  - iserve/iserve-sal-core/src/main/resources/config.properties
  - With the following contents:

```
#####################################
# Information details DO NOT MODIFY
iserve.version = ${pom.version}
#####################################

# Proxy Configuration
# http.proxyHost = proxy.company.com
# http.proxyPort = 80

# iServe Configuration
iserve.url = http://localhost:9090/iserve

iserve.services.repository = ds
iserve.services.sparql.query = http://localhost:3030/ds/query
iserve.services.sparql.update = http://localhost:3030/ds/update
```

```
iserve.services.sparql.service = http://localhost:3030/ds/data

# Documents Server
# It is recommended that you use absolute paths for storing documents as these may be deleted when redeploying the server otherwise
iserve.documents.folder = file:///tmp/iserve/service-docs/

# Logs Server
log.server = http://localhost:3030/ds/data
log.repository = ds
lufURL = http://soa4all.isoco.net/luf
```

- Compile and install
  - Execute the following commands in sequence:
    - mvn compile (this step could be skipped)
    - mvn -DskipTests=true install
- You can also generate iServe's documentation
  - While in the 'iserve' subdirectory
  - Execute 'mvn site:site'
    - This will create the documentation in the subdirectory: target/site/
  - Execute 'mvn site:run' to view the documentation locally at http://localhost:10000
- Testing
  - To run the iserve tests you need to have a web-server running. It should be configured so the document root points to
    - ./iserve-test-resources/src/main/resources/OWLS-TC4_PDDL/htdocs/
  - The web server should serve documents over port 80
  - Run the web server
  - To run iserve tests, execute 'mvn test'

Appendix B – Improvements in iServe

This new version of iServe includes a considerable range of improvements both in terms of functionality as well as from an engineering perspective towards better modularity and extensibility. We here list some of the main changes; the reader is referred to the RELEASE-NOTE files included with the code for a detailed account of these.

Functionality Improvements

- Significant performance improvements both for Services manipulation as well as Concept matching
- Pluggable Scoring and Ranking mechanism implemented for better results ordering
- Implemented new GUI based on ELDA for visualising and querying the registry
- Discovery results now include details on the inner matches for debugging and better scoring purposes
- Provisioning of Read/Write Linked API for service descriptions
- Included a notification mechanism as part of the infrastructure to support updates when changes on the underlying data occur. The implementation is based on Guava's event bus
- Added Knowledge Base manager (implementation with local indexing and/or remote SPARQL service)
- Added Ontology Crawling mechanism that automatically obtain required ontologies
- Added roll-back features for dealing with partial imports

Structural Changes

- The system has been ported to Jena as the base library for RDF and SPARQL
- Developed a new in-memory Java model and the corresponding parsers and writers for easier management of services data
- RDF Stores have been decoupled from the software by using SPARQL Updates and the Graph Management protocol instead of direct access to concrete registry implementations.
- Data management has been adapted and makes further use of named graphs for better data management and performance
- iServe now makes use of dependency injection and automated implementation loading to provide further modularity and extensibility
- Import mechanisms now based on plugins that can be loaded at runtime
- Discovery mechanisms are now pluggable and detected automatically by the system