SEAMLESS

Project IST-FP6-026476 SEAMLESS
"Small Enterprises Accessing the Electronic Market of the Enlarged Europe by a Smart Service Infrastructure"

STREP – Information Society Technologies (IST)

Deliverable D3.3.2

Distributed Storage Technical Specification

Workpackage WP3 – Technological Infrastructure
Task T3.3 – Distributed storage system

Abstract

The purpose of this deliverable and the associated task is to specify the technical framework for the SEAMLESS architecture based upon the functional specification of deliverable. This deliverable will then form the primary input for the implementation activity of SEAMLESS. As a technical basis for the implementation, this document will give a detailed insight on how SEAMLESS will be realised and on how the different components are structures and combined. This document therefore contains a set of method- and parameter descriptions.

Start date of project	Jan 1st, 2006	**Duration of project**	30 months
Deliverable due date	Jul 21st, 2007	**Actual submission date**	Jul 27th, 2007
Dissemination level	PU	**Revision status**	Final
Responsible partner	TIE	**Authors**	T3.3 participants

Table of contents

1 EXECUTIVE SUMMARY 5
2 INTRODUCTION 6
2.1 Introduction to SEAMLESS 6
2.2 Introduction to this document 6
2.2.1 Scope 6
2.2.2 Reference documents 7
3 FUNCTIONAL SPECIFICATION RECAP 8
3.1 SRRN reminder 8
3.2 SRRN changes 9
3.3 List of improvements and extensions 11
4 TECHNICAL DESIGN OVERVIEW 12
4.1 Application Layer 12
4.2 Service Layer 12
4.3 Distribution Layer 13
4.4 Repository Layer 14
5 FUNCTIONALITIES FOR IMPROVEMENT 16
5.1 Specialised ontology aspects – Query Adapter 16
5.2 Performance and response time 16
5.2.1 Functional description 16
5.2.2 Major design decisions 16
5.2.3 High level description 16
5.2.4 Service Layer 17
5.2.5 Distribution Layer 17
5.2.6 Repository Layer 18
5.3 SRRN exposure and usability 19
5.3.1 Functional description 19
5.3.2 Major design decisions 19
5.3.3 High level description 19
5.3.4 Application Layer 20
5.3.5 Services Layer 21
5.3.6 Distribution Layer. 26
5.3.7 Repository Layer 26
5.4 Node baring 27
5.4.1 Functional description 27
5.4.2 Major design decisions 27
5.4.3 High level description 27
5.4.4 Service Layer 28
5.4.5 Distribution Layer 30
5.4.6 Repository Layer 30
5.5 Selection of nodes 31
5.5.1 Functional description 31
5.5.2 Major design decisions 31

Project co-funded by the European Commission within the
Sixth Framework Programme, Information Society Technologies (IST)

5.5.3 High level description ... 31
5.5.4 Service Layer ... 31
5.5.5 Distribution Layer ... 32
5.5.6 Repository Layer ... 32
5.6 Megaping and rating ... 33
5.6.1 Functional description ... 33
5.6.2 Major design decisions ... 33
5.6.3 High level description ... 34
5.6.4 Service Layer ... 35
5.6.5 Distribution Layer ... 35
5.6.6 Repository Layer ... 35
5.7 Enhanced notification ... 35
5.7.1 Functional description ... 35
5.7.2 Major design decisions ... 36
5.7.3 High level description ... 36
5.7.4 Application layer ... 37
5.7.5 Service Layer ... 38
5.7.6 Distribution Layer ... 40
5.7.7 Repository Layer ... 40
5.8 Access information and entry monitoring ... 40
5.8.1 Functional description ... 40
5.8.2 Major design decisions ... 40
5.8.3 High level description ... 41
5.8.4 Application layer ... 41
5.8.5 Service Layer ... 42
5.8.6 Distribution Layer ... 42
5.8.7 Repository Layer ... 43
5.9 Enhanced auditing ... 44
5.9.1 Functional description ... 44
5.9.2 Major design decisions ... 44
5.9.3 High level description ... 44
5.9.4 Application layer ... 45
5.9.5 Service Layer ... 46
5.9.6 Distribution Layer ... 47
5.9.7 Repository Layer ... 47
5.10 Security 48
5.11 Content retrieval ... 48
5.11.1 Functional description ... 48
5.11.2 Major design decisions ... 48
5.11.3 High level description ... 48
5.11.4 Service Layer ... 48
5.11.5 Distribution Layer ... 49

6 SRRN LAYERS INTERFACES DESCRIPTION ... 50

6.1 Services Layer ... 50
6.1.1 Object Lyfecycle Services Interface ... 50
6.1.2 Query Services Inteface ... 52

6.1.3 Subscription Services Interface ... 62
6.1.4 Application Service Interface ... 64
6.2 Distribution Layer ... 65
6.2.1 Controller component ... 65
6.2.2 Repository-connector component ... 65
6.2.3 Registry-connector component ... 69
6.2.4 Distributor component ... 69
6.2.5 Configuration component ... 69
6.2.6 Interface to Service Layer ... 70
6.2.7 Interface to other Distribution Layers ... 74
6.3 Repository Layer ... 74
6.3.1 Controller component ... 74
6.3.2 RDF composer / decomposer ... 74
6.3.3 SPARQL to SQL converter ... 76
6.3.4 File storage adapter ... 76
6.3.5 Configuration component ... 78
6.3.6 Interface to Distribution Layer ... 78
6.3.7 WSDL ... 83
7 CONCLUSIONS ... 84

1 Executive Summary

This document is the functional specification for the distributed storage system which is to be used as a central system within SEAMLESS. The storage system is based upon the EU SEEMseed project's results and as such this document details only those improvements, fixes and enhancements necessary to move those results from the embryonic prototype of SEEMseed to a more functionality complete system and one which is capable of servicing the business needs of SEAMLESS.

This deliverable should be read in conjunction with the SEAMLESS functional specification. It describes the functions selected in the functional specification from a technical view. It therefore defines all interfaces needed by developers and software architects to design implement and run systems that are compatible to the SEAMLESS storage system. This deliverable will give some technical details in a programming language independent way. This is important in order to ensure the extensibility of SEAMLESS and in order to clarify the openness of the system.

As described in the functional specification, this document will give a detailed technical overview about the following improvements of the SEEMseed SRRN:

1. Specialised Ontology Aspects
2. Performance and Response Time
3. SRRN exposure and Usability
4. Node Baring
5. Selection of Nodes
6. Megaping and Rating
7. Enhanced Notification
8. Mapping and Content Extraction
9. Access information and Entry monitoring
10. Enhanced Auditing
11. Replication
12. Security
13. Aggregation
14. Meta Data
15. DBE integration

Each of those descriptions is performed by starting with an overview and by afterwards describing the changes necessary in all existing layers: Application layer, Service Layer, Distribution Layer and Repository Layer. - In order to give a better overview about the changes, this document will also contain an overall component interface description at section 6.

2 Introduction

2.1 Introduction to SEAMLESS

The SEAMLESS project studies, develops and experiments an embryo of the Single European Electronic Market (SEEM) network where a number of eRegistries are started up in different countries and sectors.

The main project activities are devoted to define a collaboration framework and proper business models, realise evolving sectoral ontologies, develop a technological infrastructure and a number of applications and services on top of it. Six eRegistries are installed, in Poland and Slovenia (B&C sector), in Spain, Slovakia and Romania (TEX sector), and in Hungary (generic).

The SEAMLESS project intends to provide an independent contribution to the Digital Ecosystem initiative and strictly collaborate with the relative cluster of projects.

Within the SEAMLESS project, the distributed storage system is the basis for all registry based applications. It allows building a flexible and distributed approach to realize the specific needs of all sectors. At the same time it offers a flexible and generic basis that can later be applied for other sectors. Hence, the distributed storage system is a key component in the SEAMLESS concept and it's especially important to fulfil all requirements needed for a successful infrastructure. In order to reuse proven technology and latest research results, the SEAMLESS distributed storage system is based on the SEEMseed registry and repository system (SRRN) but it extends its infrastructure with new functionality and also improves existing functionality. This is necessary in order to make the SEEMseed embryo usable in the SEAMLESS domain. Reusing SEEMseed technology allows SEAMLESS to quickly deliver a very powerful environment and to concentrate on the necessary extensions and improvements.

Within the distributed storage system, the technical specification provides a detailed technical description of the functionality, identified in the functional specification. It defines all interfaces needed by developers and software architects to design implement and run systems that are compatible to the SEAMLESS storage system. The technical specification is the most technical deliverable of the SEAMLESS project. In case that an even more detailed technical description is needed, an API specification / source code documentation is provided by the project as an informal deliverable. Furthermore, all source code is well documented with latest code documentation guidelines.

2.2 Introduction to this document

2.2.1 Scope

As defined in the SEAMLESS Description of Work (DOW), the purpose of the technical design specification document is to progress the functional specification forward to ensure that the total scope of both specifications cover for the SEAMLESS implementation:

- Functional building blocks — Functional Specification
- Design time environment — **Technical Specification**
- Run time — **Technical Specification**
- APIs and Interfaces — **Technical Specification**
- Subcomponent specifications — **Technical Specification**
- Test Scenarios — Functional Specification

The detailed design of the individual components (procedures etc) will be conducted under the individual implementation tasks and in most cases through self documentation of the code.

Project co-funded by the European Commission within the
Sixth Framework Programme, Information Society Technologies (IST)

It is of paramount importance that this document and the functional specification are read in close conjunction with each other since one is, in effect, an extension of the other. In particular the focus of this document is in taking the building blocks of the functional specification layers and extending them with the appropriate technical details with a focus on component and interface definitions.

Please note that this document – as well as the whole distributed storage system – is based on the SEEMseed SRRN implementation. Hence, this document describes only the extensions and improvements. It is therefore recommended to read the SEEMseed technical specification (D2.2.2) before reading this document in order to get a better understanding of the basic concepts and the existing implementation.

The precise sequence and scope of this document is as follows:

Section	Description
Chapter 1	Introduction
Chapter 2	Summary of the functional specifications
Chapter 3	Technical overview of the layers
Chapter 4	Detailed technical specifications of the improvement points
Chapter 5	Interface descriptions of all layers

2.2.2 Reference documents

This document is based on other deliverables of the SEAMLESS and the SEEMseed project. The following documents have been used as a base for the specifications in this deliverable:

- Formal deliverable D.3.1 SEAMLESS Overall Architecture Design
- Formal deliverable D.2.2.1 SEEMseed Functional Specification
- Formal deliverable D.3.3.a SEAMLESS Functional Specification
- Formal Deliverable D.3.3.b SEAMLESS Technological Infrastructure and its relationship to the SRRN
- Formal deliverable D.2.2.2 SEEMseed Technical Specification

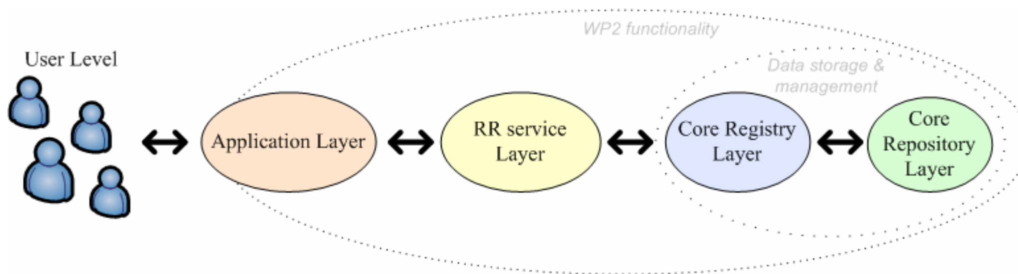
3 Functional specification recap

3.1 SRRN reminder

The SEEMseed Registry and Repository Network (SRRN) is an embryonic development which took place during the SEEMseed project. SEEM is the Single European Electronic Market and its vision is towards an Internet-based structured market where companies can collaborate without geographical and technological restraints, thus overcoming the limits of the hundreds of business services, vertical portals and private solutions each adopting its own specific model.

By creating a self-organising federated network of Repositories companies can classify their own profiles, offers and features so as to gain public visibility to potential customers and partners.

The platform is intended to ensure the required support to access, register, communicate, collaborate, and exchange information according to a peer-to-peer model. This is referred to as the SEEM Registry and Repository Network – SSRN – and is composed as shown in the figure below:



SEEMseed Layers

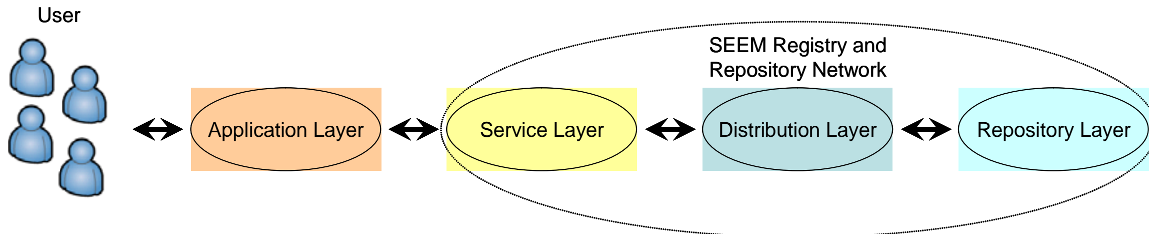
- Application Layer: This is a representation of all SEEM using applications and in this case the subset SEAMLESS application such as the ontology and application services (Search, Collaboration etc). They typically contain applications orientated and/or user orientated interfaces.
- Service Layer: This is responsible for managing the registry logic that is needed to service the applications in the most efficient way
- Distribution Layer: This enables the distribution data and the subsequent accessing of it from locations which, as a most generic principal, or invisible to the applications
- Repository Layer: The Repository Layer is responsible for storing/retrieving SRRN entries, metadata and attachments (eg documents) and for retrieving them

It should be noted that the fundamental aspects of the SRRN, noted below, should not change:

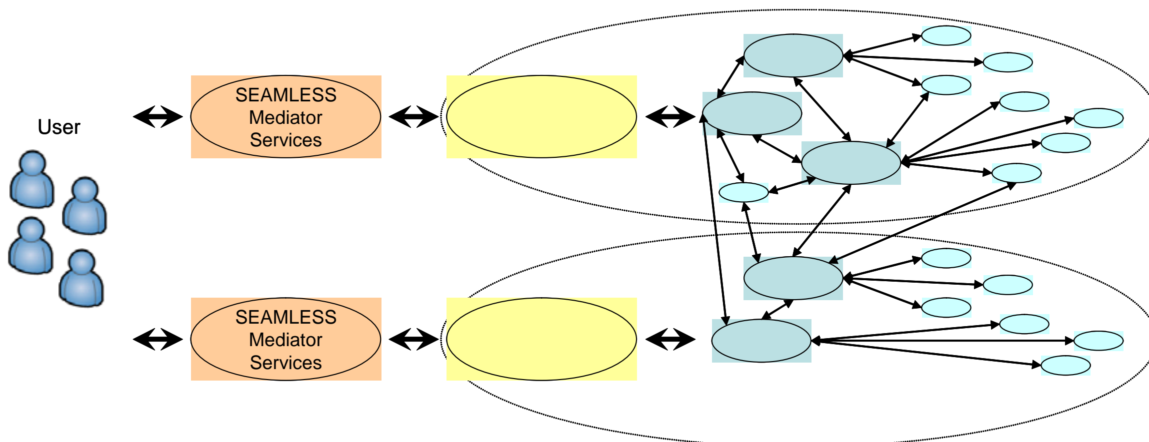
- A layered, federated P2P approach
- Support of open standards and specifications
- User and Node security, including certificate and signing support, as well as access control
- Content neutrality
- An extensible environment which can support other eBusiness layers

3.2 SRRN changes

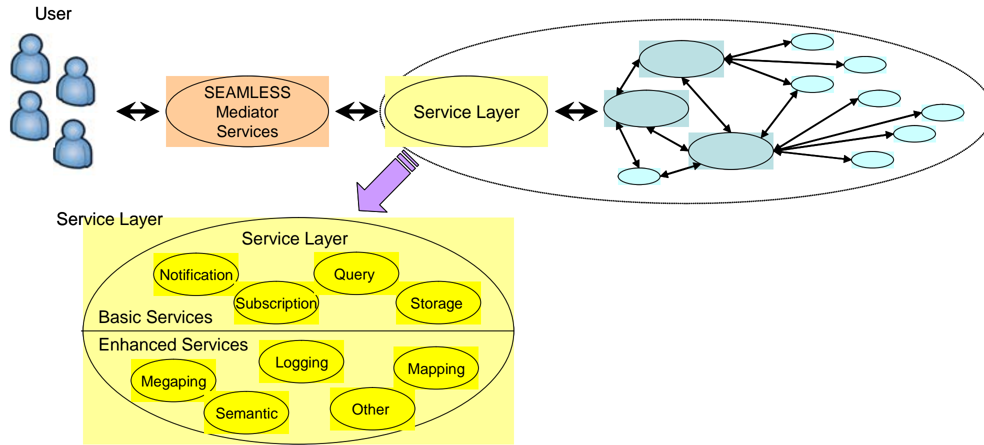
Considering first a high level perspective, it is proposed that there are several terminological and other changes made to the SRRN architecture as described in SEEMseed and indeed were on the 'to do' list of that project. These changes are as detailed below. The changes and wordings are then reflected in all subsequent texts in this document and also throughout the primary SEEM SRRN functional and technical specifications.



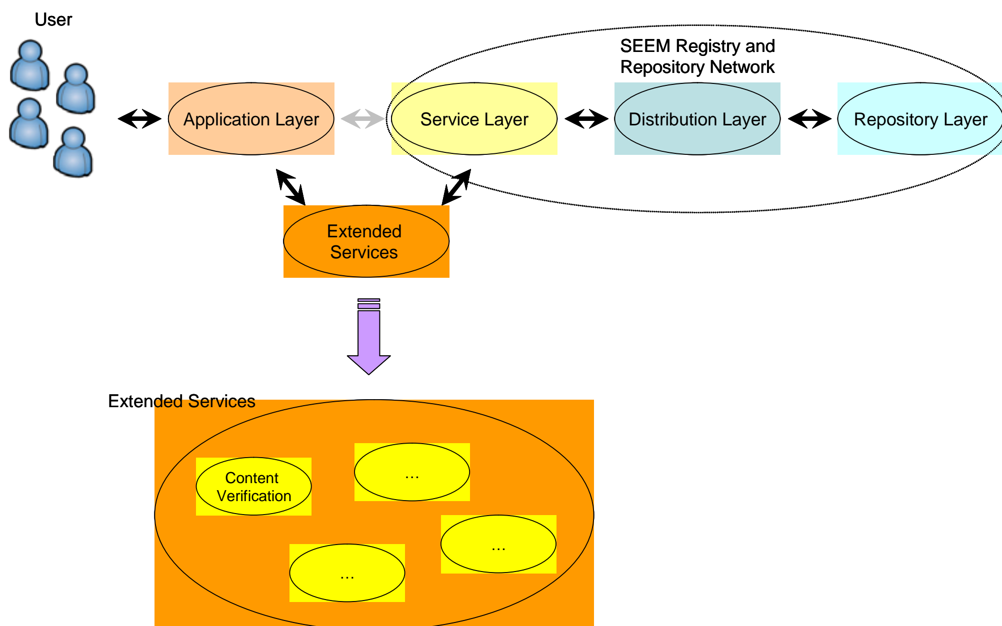
- Rebranding of the 'Core Repository Layer' to, more simply, 'Repository Layer'. Cosmetic change only
- Rebranding of the 'Core Registry Layer' to the 'Distribution Layer' since it reflects the primary reality of this layer – i.e. the distribution of select and insert queries around the repository network
- Rebranding of the 'RR Service Layer' to the 'Service' Layer. On the one hand this is a cosmetic change since the 'RR' terminology is confusing and on the other hand this adapts to the fact that the Service Layer is being adapted with additional 'enhanced' services which act on the network as a whole such as Semantic Services (from the ontology services of SEAMLESS), MegaPing Services from SEEMseed etc



- Precision that only one Distribution node can be connected at one time to a Service node. A service node may attach to additional nodes for back up/resilience etc but at run time only one node can be connected. If this principle is not adopted then it infers that Services nodes will have to perform additional actions (such as aggregation) which currently operate at the Distribution level and the duplication is contrary to the architectural modal



- Clarity that the Service Layer is composed of two types of Service:
  - Basic Services: These are elemental services which tend to operate at a data level with either little 'business' intelligence, providing aggregating functionality of atomic services and most often would provide the atomic interaction with a specific repository entry (when found/stored)
  - Enhanced Services: There have more composed functionality and are most often either:
    - Working on the SRRN nodes as a whole – it is simple that implementation via the Service Layer is 'logical' since it is the entry point. The alternative would be trying to specify it as an additional 'Application' (in fact and 'Extended Service – see later) although the difference between the two is a 'grey' area. Examples include MegaPing, and MegaLog
    - Provide upgraded services to an application to minimise the application functionality that might otherwise need to be created. These typically do not work with other layers but simply provide a more complex operation at needed in the Service Layer itself. Examples include Semantic and Mapping Services



- Addition of an Extended Service Layer which can be invoked if necessary by the application layer and acts as a bridge to the SRRN Service Layer. Extended Services are in a grey area: on the one hand the services they offer are relatively complex making use of the SRRN features as if the extended service itself was another application (for example a content checking service); on the other hand the extended service has no value on its own and it must be used by a business application (for example to trigger the checking of content) and thus it might have been classed as an enhanced service in the Service Layer. One possible factor to decide between the two is that when a new service function is needed an 'extended service' is built to fulfil it. If it becomes clear that many people require this function then it is probably time to consider whether it should more intrinsically in the SRRN itself as a enhanced service.

3.3 List of improvements and extensions

Within the functional specification, the following list of improvements and changes to the SEEMseed SRRN has been identified:
In this document all changes will be described in an own section. Each section will be broken down into its influences to the different layers:

- Specialised Ontology Aspects
- Performance and Response Time
- SRRN exposure and Usability
- Node Baring
- Selection of Nodes
- Megaping and Rating
- Access information and Entry monitoring
- Enhanced Notification
- Mapping and Content Extraction
- Access information and Entry monitoring
- Enhanced Auditing
- Replication
- Security
- Aggregation
- Meta Data
- DBE integration

At the end of this document, a summary about the resulting interfaces will be given.

4 Technical design overview

4.1 Application Layer

The application layer consists of the set of application developed in WP4. Query, negotiation and other applications will be developed that will make use of the distributed storage system.

4.2 Service Layer

The following diagram shows the overall component and interface model of the Service Layer. The Service layer is an interface that allows the applications to access to the information registered and stored into the registry in a homogeneous way. The scope of the functionality of the Service Layer is constrained by the possibilities that the Distribution Layer offers to it, and by the information model offered by the core itself.

Main functions of the Service Layer are the management of the object storing and retrieval functionality, including automated metadata extraction and consistency check, high-level functionality for querying the registry and the subscription of information covering the definition and the notification mechanisms.

Applications are built on top of the Service Layer. The interfaces between Service Layer and the applications are implemented by Web Services. The Service Layer offers three main Web Service interfaces to the Application Layer:

- Object Lifecycle Services Interface
- Query Services Interface
- Subscription Services Interface

The following figure provides an overview of the main components of the Service Layer and the data flow between the components. The following sections of this document specify the components on a technical level.

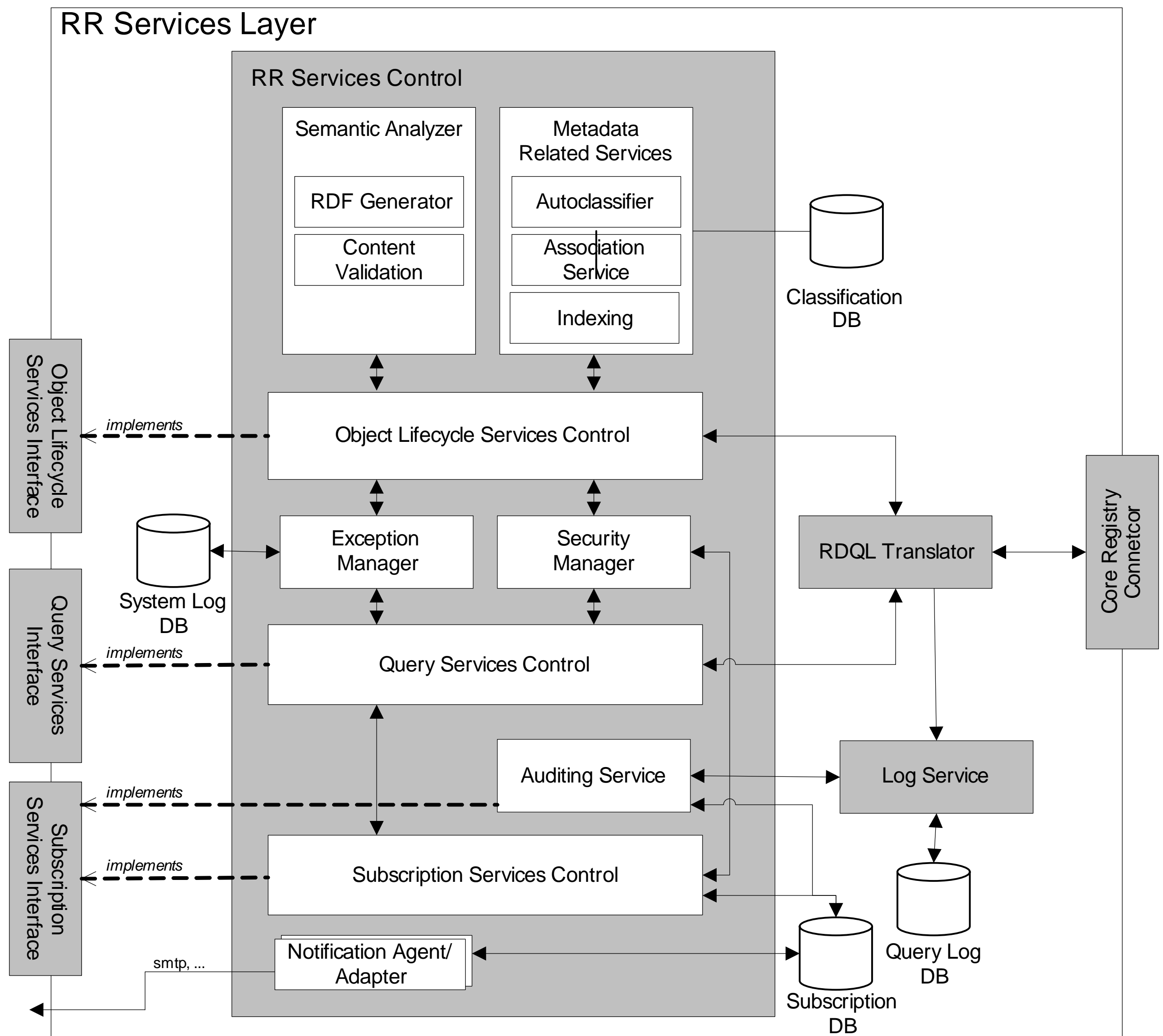


Figure 1: Component overview Service Layer

4.3 Distribution Layer

The following diagram shows the overall component and interface model of the Distribution Layer.

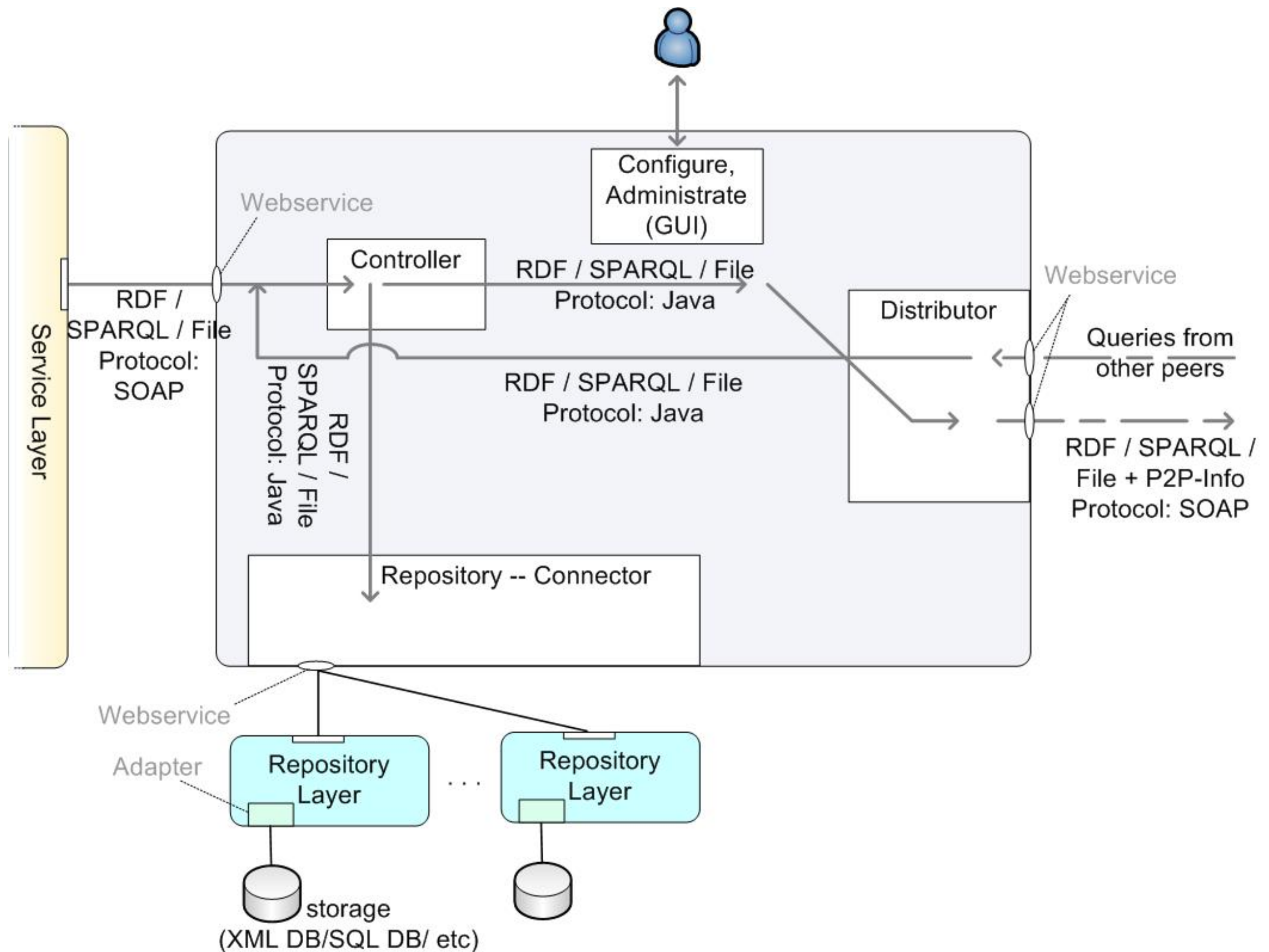


Figure 2: Component and Interface Model

The Distribution Layer is connected to the Service Layer, which uses webServices to send files and metadata. This data is accepted by a webService and forwarded to a Controller-component, which cares about the distribution of the data and the control of the overall process. The Controller sends the queries/data to one or more of three components:

1. The Repository-Connector: To connect to repositories that this specific registry is aware of.

2. The Distributor: To connect to the full network of registries

4.4 Repository Layer

The following diagram shows the overall component and interface model of the Repository Layer. These are described in detail in section 6.

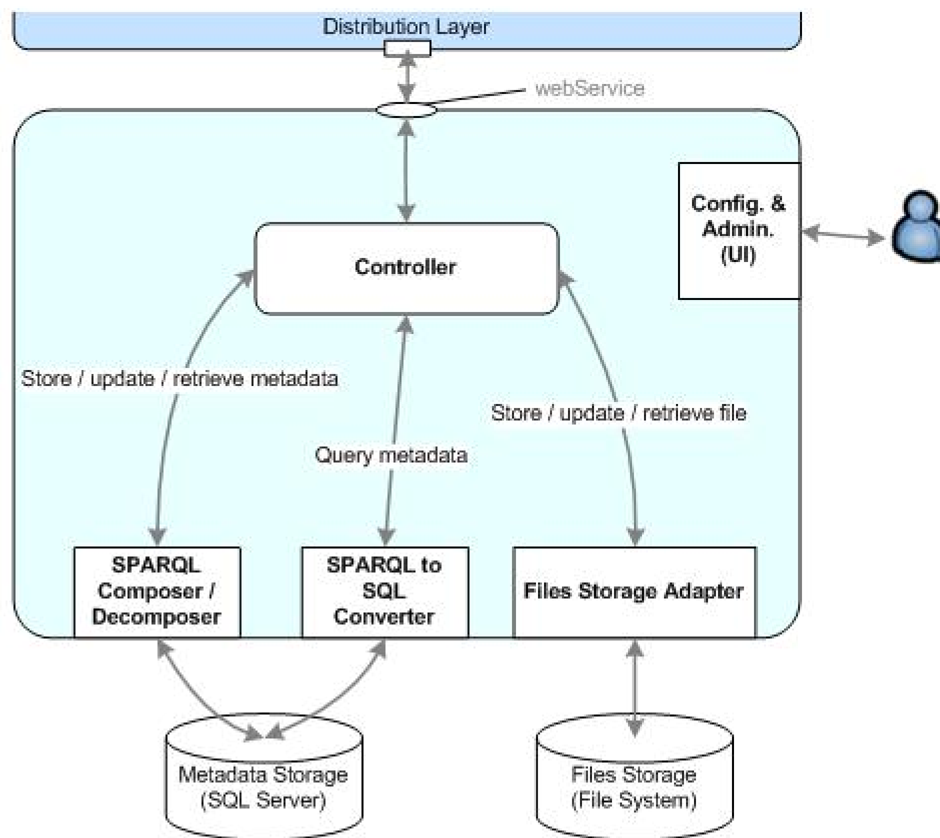


Figure 3: Repository Layer – Component and Interface Model

The Repository Layer is connected to the Distribution Layer via a webService interface. These calls are handled by the *Controller* component, which is solving also the security related issues, and after that takes care of the distribution of the data, and the control of the overall process. The Controller sends the queries/data to other internal components, depending on the type of data that has to be handled.

5 Functionalities for improvement

5.1 Specialised ontology aspects – Query Adapter

This functionality is described in the Query Adapter Functional and Technical Description document of WP2.

5.2 Performance and response time

5.2.1 Functional description

The main performance drawback found from experiences in the SEEMseed project is the synchronous communication protocol used between the Service Layer and the Distribution Layer. The Distribution Layer already has an asynchronous communication interface implemented but this interface is not used by the Service Layer.
There seems to be a time consuming communication problem between the layers. SEAMLESS will take care of this by refactoring the communication between the layers and by using an advanced Ping technology the enables us to find timing problems.

5.2.2 Major design decisions

- All query functions will use a timestamp. If a layer receives a query with an expired timestamp then the layer should ignore the query.
- The timestamp that will be used is the Unix timestamp mechanism. This timestamp mechanism also takes into account the difference between time zones. The timestamp is the number of seconds since 1st of January 1970 UTC time.
- All nodes in the SEAMLESS network should be connected to a time server in order to synchronize the clock of all nodes. This is necessary to make sure the timestamp will function correctly.

5.2.3 High level description

The main performance improvements should be gained by improving the asynchronous interface between the Service Layer and the Distribution Layer.

The timestamp mechanism should enforce that queries are not executed anymore after it has expired. By forcing this valuable processor and user time are not wasted anymore one getting query results that are later ignored on the application layer.
In order to find any time leak in the SRRN a test query interface will be created to test for any leaks. This QueryTest interface will be created in all the layers. This Query Test should go from the application Layer to a single Repository node and all the way back to the Application Layer. The test query should not be distributed but only go to one repository node. The repository should not perform any database transaction but just return the query.
An XML structure will be passed where nodes should store the time when they processed the query.

```
<SEEMQueryTest>
    <Node>
        <SEEMID>seem-12</SEEMID>
        <NodeType>Service</NodeType>
        <Time>12:54:34</Time>
    </Node>
    <Node>
        <SEEMID>seem-16</SEEMID>
        <NodeType>Distribution</NodeType>
        <Time>12:54:35</Time>
```

```
        </Node>
        <Node>
                <SEEMID>seem-11</SEEMID>
                <NodeType>Repository</NodeType>
                <Time>12:54:35</Time>
        </Node>
</SEEMQueryTest>
```

5.2.4 Service Layer

5.2.4.1 Major design decisions

No specific considerations at this layer.

5.2.4.2 High level description

A QueryTest function has to be added that will receive a test query and will pass the query to one distribution node.

5.2.4.3 Component interface description

In the QueryAsync method the timeout will be removed and instead a timestamp will be included in the parameter list.

For the QueryTest the following function will be added to the Query Services Interface:

string [] queryTest (String xmlQueryTest)
The method performs a QueryTest through all the layers and returns an XML structure with the test results.
Parameters:
string xmlQueryTest – This function expects an XML structure with the current test results.
Return value:
string - The return value of the query is an XML structure with the results of the test query.
Exceptions:
exception - Throws a queryError exception in case of an error

5.2.5 Distribution Layer

5.2.5.1 Major design decisions

No specific considerations at this layer.

5.2.5.2 High level description

A QueryTest function has to be added that will receive a test query and pass the query to one repository node. The query should only be passed to one repository node and should not be distributed to other distribution nodes.

5.2.5.3 Component interface description

In the queryAsync method a timestamp will be included in the parameter list to indicate the time after which the query should not be executed anymore.

For the QueryTest the following functions will be created in the Distribution Layer repository interface:

string queryTest (string xmlQueryTest) throws Exception

Performs a test query on one connected repository.
This function is used to perform a test query on a single repository to identify any problem in the communication between all layers/nodes. The Distribution Layer should not distribute the query to other distribution nodes but just pass the query to one connected repository node.

Parameters:

xmlQueryTest – xml document with data on all passed nodes

Returns:

XML document with data on all passed nodes

Exceptions:

Error - Throws an exception in case of an error

5.2.6 Repository Layer

5.2.6.1 Major design decisions

- The transaction database will be an XML database maintained by the Repository Layer.

5.2.6.2 High level description

A QueryTest function has to be added that will receive a test query from a distribution node. The Repository Layer will not perform a database transaction but just add the timestamp information to the testquery xml structure and will return the call to the distribution node.

Because multiple Distribution Layers could be connected to the same repository it is possible that the repository will receive the same query multiple times. In order to not waist valuable processor and user time the query should be executed only once and subsequent calls of the same query should not be executed. The repository will maintain a database with recently executed queries and when a query is received it should first check this database before the query is executed.
In order the keep transactions to this database fast the Repository Layer will have a cleaning service that will remove transaction records that have expired.

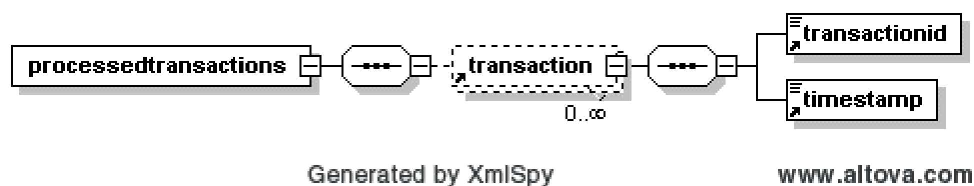


Figure 4: XML schema for transaction database

Example XML database:

```
<pocessedtansactions>
        <transaction>
                <transactionid>c3adaab0-759f-11db-9fe1-0800200c9a66</transactionid>
                <timestamp>1163698174</timestamp>
        </transaction>
        <transaction>
                <transactionid>e4b30930-759f-11db-9fe1-0800200c9a66</transactionid>
                <timestamp>1163701779</timestamp>
        </transaction>
```

</pocessedtansactions>

5.2.6.3 Component interface description

A timestamp parameter will be added to the query methods.

A function QueryTest will be added to the Interface to the Distribution Layer:

string queryTest (string xmlQueryTest)

Performs a test query on the repository.
This function is used to perform a test query on a single repository to identify any problem in the communication between all layers/nodes. The Repository Layer doesn't perform any query to the database but update the xmlQueryTest structure and return to the Distribution Layer.

Parameters:

xmlQueryTest – xml document with data on all passed nodes

Returns:

XML document with data on all passed nodes

Exceptions:

Error - Throws an exception in case of an error

5.3 SRRN exposure and usability

5.3.1 Functional description

Some new utilities will be offered in the Service Layer to improve the handling of documents and versions. Methods for retrieving documents by the enveloped metadata seem_id, retrieve latest version of a document or metadata, getting the identifiers of all existing versions of a document or metadata are added.

Also is added the support to execute queries with attribute values in the select clause of the SPARQLquery in the synchronous and asynchronous queries so that an application can retrieve, as a result of a query, a set of values from the matching metadata instead of a set of metadata that has to be parsed to extract the desired attribute values.

A new feature is the support for pluggable services in the Service Layer so that applications can share or export their own services to others or enhancements over the existing ones.

5.3.2 Major design decisions

Support for execute queries with attribute values in the select clause of the SPARQLquery in the synchronous and asynchronous queries will be added so that an application can retrieve as a result of a query a set of values from the matching metadata instead of a set of metadata that has to parse after to extract the desired attribute values. Applications could get metadata values from SPARQLqueries without parsing the metadata.

As a new feature the support for pluggable services in the Service Layer is added so that applications can share or export their own services to others or enhancements over the existing ones.

5.3.3 High level description

No changes in the interface of the Distribution neither Repository Layer are needed but the support for return attributes as a result of SPARQLqueries with attributes in the select clause. All other new utilities or improvements over the existing ones involve only changes in the

Service Layer implementation.

1. Queries that return attributes instead of seemID or metadata must be supported in both synchronous and asynchronous queries. These queries will return an xml document. There isn't a xml schema for the return value of the query so the xml document composition depends on the name and the number or attributes selected, is a variant xml document which consists in a node list of "results" nodes in which each result node have a child node that corresponds to every selected value in the query and each child node text corresponds to the returned value in the query.

For example, for a SPARQLquery that selects the name and the address from the metadata:

```
select ?name, ?address where ... <some filtering criteria>
```

The execution result will be an xml with the following structure:

```
<results>
    <result>
        <name>Juan Perez Martinez</name>
        <address>Sagunto, 13</address>
    </result>
    <result>
        <name>Paco Maroto</name>
        <address>Hierba, 1</address>
    </result>
</results>
```

2 .The services layer will allow deploying pluggable services for applications with some limitations. Not all existing programming languages are supported and not all the web services implementations are supported. The Service Layer will provide a Jakarta Axis for Java and a Jakarta Tomcat for deploying the web services, so the pluggable service implementation must be in Java and all its library dependencies must be included in service code package (except packages contained in the standard j2se distribution, tomcat distribution and Axis distribution). So for the deployment of a pluggable service a java package with the service implementation and a WSDL file are needed.

The lifecycle of pluggable services as well as aspects related to access control, service security, etc are managed by the own pluggable service under the responsibility of the applications. The Service Layer only provides a shared execution environment.

5.3.4 Application Layer

5.3.4.1 Major design decisions

No specific considerations at this layer

5.3.4.2 High level description

No specific considerations at this layer.

5.3.4.3 Component interface description

No specific considerations at this layer.

5.3.5 Services Layer

5.3.5.1 Major design decisions

Some changes in the existing Service Layer interface have been added:

- Methods that create a document or metadata add the initial access rights for the new created object in the repository.
- Queries with attributes in the select clause are supported.
- Query functions should pass the userid in order

New methods in the Service Layer interfaces have been added:

- To retrieve a document by the seemID of the envelop metadata
- To get the seemID of the metadata that envelops a document.
- Delete a document or a metadata and all it existing versions.
- Get the last version of a document or a metadata.
- Get all existing versions (metadata or document identifiers) of a document or metadata.

5.3.5.2 High level description

Methods to create a new document or metadata (store, storeMetadata, storeRawDocument) will include a parameter with the initial object access rights and only if that parameter is null the default access rights will be applied to the object. The access rights parameter is an xml document based on the access rights xml schema.

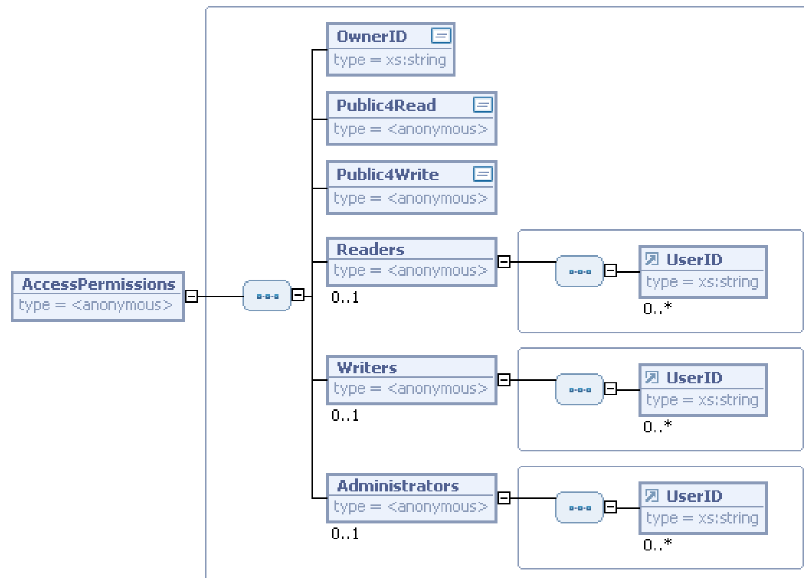


Figure 5 Access rights xml schema

The default access rights applied to a document or metadata will be:

- Owner: the object creator.
- Administrators: the object creator.
- Public read: yes.
- Public write: no.
- Readers: none.
- Writers: none.

The methods to execute synchronous and asynchronous queries in the repository will support select attribute values in the SPARQL select clause but they don't change its interface neither its implementation because the new functionality is delegated to the Distribution Layer.

A new method is created in the Service Layer interface to get a document from the repository by the seemID of the metadata that links to the document. This method act as a façade for the applications and it must get the metadata corresponding to seemID, extract from the metadata the document identifier and return the document content.

A new method is created in the Service Layer interface to get the seemID identifier of the metadata that links to a document.

A method to delete a document and all its versions is created. The method receives as input parameter a document_id and retrieves all the metadata and linked documents that are versions of the document_id. If the user has write access to every retrieved metadata and document then all metadata and all documents are removed.

A method to delete a metadata and all its versions is created. The method receives as input parameter a seem_id and retrieves all its versions metadata. If the user has write access to every metadata then the metadata and its linked documents (if any) will be removed.

A method to get the last version of a metadata is created. The method receives as input parameter a seem_id and if the user has access rights to the metadata, the seemID of the latest version of the metadata is returned.

A method to get the latest version of a document is created. The method receives as input parameter a document_id and if the user has access rights to the document, the documentID of the latest version of the document is returned.

A method to get all the versions of a document is created. The method receives as input parameter a document_id and retrieves all available document versions (minors and majors) from the repository. The result is a list of document_id for each of the documents to which the user has access rights.

A method to get all the versions of a metadata is created. The method receives as input parameter a seem_id and retrieves all available metadata versions (minors and majors) from the repository. The result is a list of seemID of each one of the metadata to which the users has access rights.

The services layer will allow the use and deployment of pluggable services for applications with some limitations. Not all existing programming languages are supported and not all the web services implementations are supported. The Service Layer will provide a Jakarta Axis for Java and a Jakarta Tomcat for deploying the web services, so the pluggable service implementation must be in Java and all its library dependencies must be included in service code package (except packages contained in the standard j2se distribution, tomcat distribution and Axis distribution). So for deploying a pluggable service a java package with the service implementation and a WSDL file is needed.

The lifecycle of pluggable services as well as aspects related to access control, service security, etc are managed by the own pluggable service under the responsibility of the applications. The Service Layer only provides a shared execution environment.

5.3.5.3 Component interface description

boolean queryAsync (String *query*, long timestamp, String *fromEndpointURL*, String *fromServiceURN*, String *transactionID*, String *userPublicKey*) throws Exception

This method performs an asynchronous search on the metadata objects stored in the registry. When this method is executed it translates the query to registry and returns the control immediately. As soon as the registry is going collect query results, the results are returned calling the queryResult or queryResultContent methods in the fromEndpointURL and fromServiceURN provided by the application layer. Works as the queryMetadata method but return results in a asynchronous way.

Parameters:

query The RDF query to execute.
timestamp Timestamp after which results should be ignored.
fromEndpointURL Endpoint from application layer where receive results
fromServiceURN Service URN from application layer where receive results
transactionID Transaction id associated to request
userPublicKey Public key of the user that execute service

Returns:

True if the request is delivered to Distribution Layer, false otherwise.

Exceptions:

Exception Throws an exception in case of an error.

byte [] getDocumentBySeemID (String transactionID, String seemID, String userPublicKey, String type) throws Exception

Method to retrieve documents from the repositories by the metadata seemID that contains the document in the repository.

Parameters:

transactionID - Transaction id associated to request
userPublicKey - Digital Signature of the user that executes the service
seemID - The identifier of the document to be retrieved
type - Type of codification in which the document will be returned. The allowed values are: MIME, DIME and BASE64 (for attachments of less than 1Mb. mainly to work from mobile devices).

Returns:

If the type equals BASE64 then the document is returned as byte array, if not, the document is included into the response following the DIME or MIME codification.

Exceptions:

Exception Throws a queryError exception in case of an error

byte [] getDocument (String transactionID, String documentID, String userPublicKey, String type) throws Exception

Method to retrieve a document from the repositories by its document_ID

Returns:

If the type equals BASE64 then the document is returned as byte array, if not, the document is included into the response following the DIME or MIME codification.

Parameters:

documentID - The document_id of the document to retrieve
transactionID - Transaction id asociated to request
userPublicKey - Digital Signature of the user that executes the service
type - Type of codification in which the document will be returned. The allowed values are: MIME, DIME and BASE64 (for attachments of less than 1Mb. mainly to work from mobile devices).

Exceptions:

Exception Throws a queryError exception in case of an error

String getMetadataIDFromDocument (String *transactionID*, String *documentID*, String *userPublicKey*) throws Exception

Method to retrieve the identifier of the metadata that envelops a document.

Returns:

The seem_id of the metadata that envelops the document in the repository

Parameters:

documentID The document_id of the document to retrieve
transactionID Transaction id associated to request
userPublicKey Digital Signature of the user that executes the service

Exceptions:

Exception Throws a queryError exception in case of an error

boolean removeAllDocumentVersions (String *transactionID*, String *documentID*, String *userPublicKey*) throws Exception

Removes all the existing versions of the document specified by document_id in the repository. All the metadata that envelop the deleted documents in the repository are deleted too.

Parameters:

transactionID Transaction id associated to request
userPublicKey The user's Digital Signature.
documentID Id of the object in the repository to be deleted

Returns:

true if success or false if error.

Exceptions:

Exception Throws a LifeCycle exception in case of an error

boolean removeAllMetadataVersions (String *transactionID*, String *seemID*, String *userPublicKey*) throws Exception

Removes all the existing versions of the metadata specified by seem_id in the repository. If the metadata deleted have documents associated, the documents are deleted too.

Returns:

true if success or false if error.

Parameters:

transactionID Transaction id asociated to request
userPublicKey The user's Digital Signature.
seemID Id of the object in the repository to be deleted

Exceptions:

Exception Throws a LifeCycle exception in case of an error

String getLastVersionDocument (String *transactionID*, String *documentID*, String *userPublicKey*) throws Exception

Get the last version of the document specified by document_id in the repository.

Returns:

The document_id of the last version of the document which identifier is documentID.

Parameters:

transactionID Transaction id associated to request
userPublicKey The user's Digital Signature.
documentID Document id from which retrieve all its versions

Exceptions:

Exception Throws a LifeCycle exception in case of an error

String getLastVersionMetadata (String *transactionID*, String *seemID*, String *userPublicKey*) throws Exception

Get the last version of the metadata specified by seem_id in the repository.

Returns:

The seem_id of the last version of the metadata which identifier is seemID.

Parameters:

transactionID Transaction id associated to request
userPublicKey The user's Digital Signature.
seemID Metadata seem id from which retrieve all its versions

Exceptions:

Exception Throws a LifeCycle exception in case of an error

String [] getAllDocumentVersions (String *transactionID*, String *document_id*, String *userPublicKey*) throws Exception

Get the document id's from all the existing versions of the document which documentID is equal to document_id in the repository.

Returns:

The document_id's of the existing versions.

Parameters:

document_id The document identifier from which retrieve all the available versions
transactionID Transaction id associated to request
userPublicKey The user's Digital Signature.

Exceptions:

Exception Throws a LifeCycle exception in case of an error

String [] getAllMetadataVersions (String *transactionID*, String *seem_id*, String *userPublicKey*) throws Exception

Get the seem id's from all the existing versions of the metadata which seemID is equal to seem_id in the repository.

Returns:

The seem_id's of the existing versions.

Parameters:

seem_id The metadata seem_id from which retrieve all versions available
transactionID Transaction id asociated to request
userPublicKey The user's Digital Signature.

Exceptions:

Exception Throws a LifeCycle exception in case of an error

String deployApplicationServices (byte *service_code_jar_binary*[], String *service_wsdl*, String *transactionID*) throws Exception

Deploy a pluggable application service extension for share with other applications.

Returns:

The service endpoint url in the Service Layer

Parameters:

transactionID Transaction identifier of the request
service_code_jar_binary Java binary .jar file with the service implementation
service_wsdl wsdl of the service to deploy

Exceptions:

Exception Error if fails the deployment

String modifyApplicationServices (byte *service_code_jar_binary*[], String *service_wsdl*, String *transactionID*) throws Exception

Modify an existing pluggable application service extension for share with other applications.

Returns:

The service endpoint url in the Service Layer

Parameters:

transactionID Transaction identifier of the request
service_code_jar_binary Java binary .jar file with the modified service implementation
service_wsdl wsdl of the modified service

Exceptions:

Exception Error if fails the deployment

boolean deleteApplicationServices (String *pluggable_service_endpoint_url*, String *transactionID*) throws Exception

Remove an existing pluggable application service extension.

Returns:

true if the service is undeployed or false if error

Parameters:

transactionID Transaction identifier of the request

pluggable_service_endpoint_url Service Endpoint of the pluggable service to remove.

Exceptions:

Exception Error if fails the undeploy

5.3.6 Distribution Layer.

5.3.6.1 Major design decisions

Must give support for queries with attributes in the select clause so that the Service Layer receives the query result in the appropriate format see paragraph 5.3.3

5.3.6.2 High level description

The query methods interface hasn't changed but some implementation changes are needed. Queries that return attributes instead of seemID or metadata must be supported in both synchronous and asynchronous queries, and will return an xml document. There isn't a xml schema for the queries return so the xml document composition depends on the name and the number or attributes selected, is a variant xml document which consists in a node list of "results" nodes in which each result node has a child node that corresponds to every selected value in the query and each child node text corresponds to the returned value in the query.

For example, for a SPARQLquery that selects the name and the address from the metadata:

select ?name, ?address where ... <some filtering criteria>

the execution result will be a xml with the next structure:

```
<results>
        <result>
                <name>Juan Perez Martinez</name>
                <address>Sagunto, 13</address>
        </result>
        <result>
                <name>Paco Maroto</name>
                <address>Hierba, 1</address>
        </result>
</results>
```

5.3.6.3 Component interface description

No changes need to be made in the interface.

5.3.7 Repository Layer

5.3.7.1 Major design decisions

Must give support for queries with attributes in the select clause so that the Distribution and Service Layer receive the query results in the appropriate format see paragraph 5.3.3

5.3.7.2 High level description

The query methods interface hasn't changed but some implementation changes are needed. Queries that return attributes instead of seemID or metadata must be supported in the

synchronous queries, and will return an xml document. There isn't a xml schema for the queries return so the xml document composition depends on the name and the number or attributes selected, is a variant xml document which consists in a node list of "results" nodes in which each result node has a child node that corresponds to every selected value in the query and each child node text corresponds to the returned value in the query.

For example, for a SPARQLquery that selects the name and the address from the metadata:

```
select ?name, ?address where ... <some filtering criteria>
```

the execution result will be a xml with the next structure:

```
<results>
        <result>
                <name>Juan Perez Martinez</name>
                <address>Sagunto, 13</address>
        </result>
        <result>
                <name>Paco Maroto</name>
                <address>Hierba, 1</address>
        </result>
</results>
```

5.3.7.3 Component interface description

No changes need to be made in the interface.

5.4 Node baring

5.4.1 Functional description

When making a search for any kind of information, usually, the query is sent to all the available nodes in order to get the results. Basically this action is performed by using a network flooding algorithm that includes all node profiles (quick and slow nodes, reliable and not reliable nodes, etc.).

The main reasons to include the node baring are:

- Even being in a global network, applications or users, will have the ability of searching e.g. only in the nodes they uses to store information.
- Allow slow or not reliable routes that can result in constrained search results.
- Avoid querying in non trusted nodes or nodes that are managed by non trusted organisations.

The changes will lead to improvements on both performance and trust.

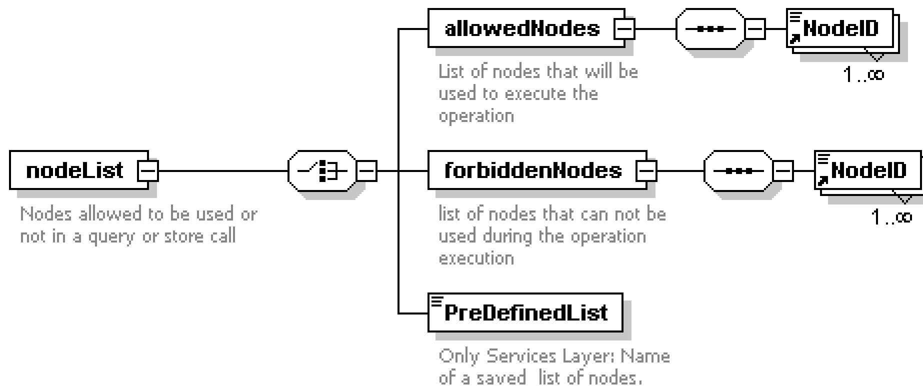
5.4.2 Major design decisions

This functionality affects to the Service and Distribution Layers of the SRRN. A modification of the query interface methods is needed, by adding a new parameter that will include the nodes allowed or forbidden for this specific call. In order to maintain backwards compatibility, an additional method will be added with this change and the rest of the modifications that come from improvements described in this document.

5.4.3 High level description

When a node of the Service or Distribution Layer receives a query call, it has to read the baring parameter and process it. Then before retransmitting this query call to the next nodes it has to check that these nodes are in the allowed or forbidden list so that it will retransmit the call to these nodes. The Services layer has to only care of Distribution nodes, but Distribution nodes have to be taking into account relaying both to other Distribution nodes and Repository nodes.

The following XML schema describes the format of this new parameter to be managed in the Service and Distribution Layer.



Allowed Nodes: Is the list of nodes that can be used for retransmitting the operation in execution to the lower layers.
Forbidden Nodes: Is the list of the nodes that can never be used for the execution of the operation being managed.
PredefinedList: Services are able of saving nodeLists associated to certain names. So for example, an administrator can save a list of nodes under the name “mySRRNNodes” and can use it by simply using this name in this parameter.

5.4.4 Service Layer

5.4.4.1 Major design decisions

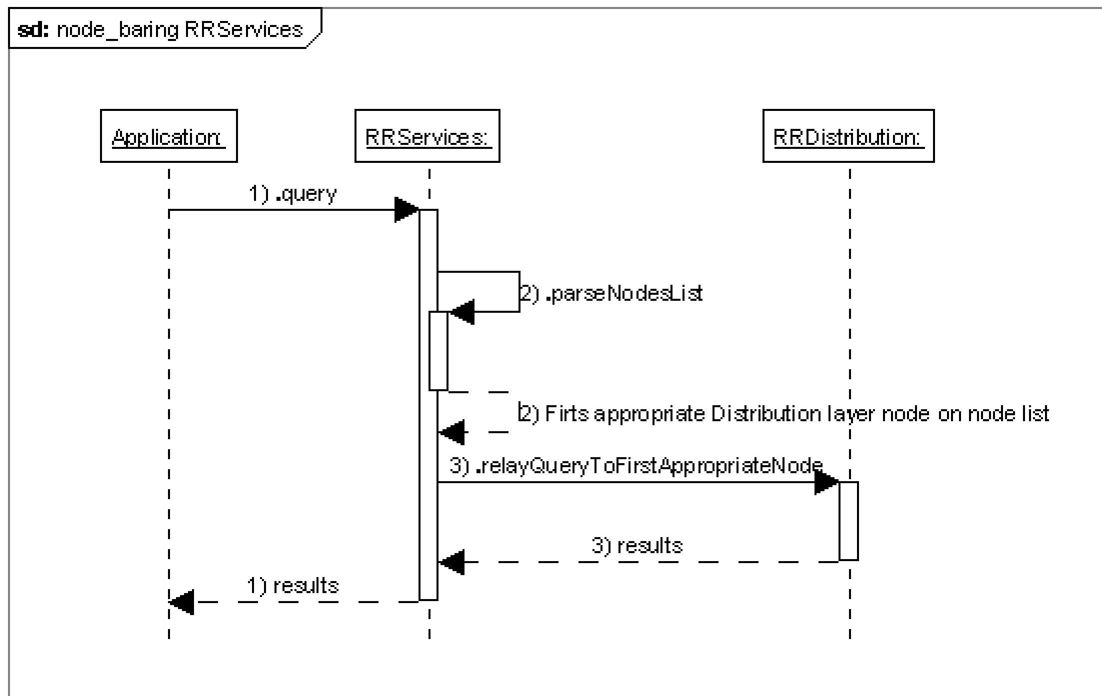
The query interface offered to the Application layer has to be updated in order to accept the list of allowed nodes. In order to maintain backwards compatibility, the new parameter will be included in a new version of each of the functions, leaving the previous version unchanged.

The lifecycle interface of the Service will be updated, adding three new access methods one for adding new predefinedLists of nodes, delete it, or update. The query interface has also to be updated with a new method to retrieve the list of available predefined node lists.

Existing interfaces to be updated
- getMetadata
- megaPing
- queryAsync
- queryMetadata

5.4.4.2 High level description

Service Layer maintains a list of the connected distribution nodes. The list of Distribution nodes is actually used as a backup in order to maintain a working system, so, the list is used sequentially, and only if the first distribution node fails, then it uses the second. So to implement this new functionality, the Service layer will parse this list sequentially in order to know if the nodes can be used to relay the queries or not.
If instead of a list of nodes, a preDefined list is passed and then a specific method has to recover the nodes description associated to this list and process the information.



For storing and retrieving predefined lists of nodes, a new database table will be added. This new table (and so the access methods) will only be managed by the administrator of the Service node.

5.4.4.3 Component interface description.

See interfaces to be updated in the final interface description.

New methods are:
boolean addPreDefinedNodeList (String listName, String description, String NodeList) throws Exception
Method that allow to add new named node lists for implementing node baring functionality
Parameters:
listName name of the node list, to be retrieved afterwards. Univocally identifies each node list.
description textual description of the list
NodeList List of nodes following the nodeList schema.
Returns:
true in case the operation is successfully executed.

Exceptions:
Exception If any error occurs, then an exception is thrown.

Boolean deletePreDefinedNodeList (String listName) throws Exception
Node baring - Method that allow to delete a named node lists.
Parameters:
listName name of the node list, to be retrieved afterwards. Univocally identifies each node list.
Returns:
true in case the operation is successfully executed.

Exceptions:
java.rmi.RemoteException If any error occurs, then an exception is thrown.

Boolean modifyPreDefinedNodeList (String listName, String description, String NodeList) throws Exception
Node Baring - Method that allow modifying a node list
Parameters:
listName name of the node list to be retrieved modified.
description modified textual description of the list
NodeList Modified list of nodes following the nodeList schema.
Returns:
true in case the operation is successfully executed.
Exceptions:
java.rmi.RemoteException If any error occurs, then an exception is thrown.

String [] getPreDefinedNodeList ( )
Function that allows retrieving the names of the stored predefined Nodes Lists.
Returns:
List of name,description pairs of the predefined nodeLists stored.

5.4.5 Distribution Layer

5.4.5.1 Major design decisions

A new parameter will be added in the query interfaces. This parameter will be the same nodeList that have to be relayed from the Service layer, in order to allow the Distribution Layer to fulfil the allowed or forbidden list of nodes sent by the application.

Existing interfaces to update:
- queryAsync
- megaping

5.4.5.2 High level description

As the Distribution Layer needs a subscription or joining process in order to allow both other distribution layers and Repository Layers to be attached to a specific distribution node, each of the distribution nodes is managing a list of its connected nodes. This means that when a Service layer send a query to the Distribution Layer, it has to parse the list of nodes sent and relay the query only to these nodes that fulfil the conditions expressed in this parameter.

5.4.5.3 Component interface description.

See interfaces to be updated in the final interface description.

5.4.6 Repository Layer

There are no implications in the Repository Layer.

5.4.6.1 Major design decisions

There are no implications in the Repository Layer.

5.4.6.2 High level description

There are no implications in the Repository Layer.

5.4.6.3 Component interface description

There are no implications in the Repository Layer.

5.5 Selection of nodes

5.5.1 Functional description

When a user is going to store or version any kind of data, he can add additional parameters to the storage calls in order to specify the SEEMIDs of the repositories where he wants to store the data.

If some repository SEEMID is specified, then the SRRN will try to store the data in the first repository of the list, and if something goes wrong, then it will try in the next repositories, so the list implies preference.

In case there is only a specified a repository, the data must be stored in this repository.

So, in case the repository is not available, the SRRN will throw an exception and an error will be returned to the user. In case of none of the repositories in the repository list are available, and able to store the information (due to security and/or access rights policies) then an exception and error will be also thrown.

5.5.2 Major design decisions

This functionality affects to the Service and Distribution Layers of the SRRN. A modification of the query interface methods is needed, by adding a new parameter that will include the nodes allowed or forbidden for this specific call. In order to maintain backwards compatibility, an additional method will be added with this change and the rest of the modifications that comes from improvements described in this document.

5.5.3 High level description

The way of working is similar to the node baring functionality, and it affects also to the Service and Distribution Layer. The same XML schema can be used to specify preference in the repository to be used to store data. In this case, if some data is specified in the forbidden nodes field, it will be discarded. The Service will send the first (preferred) repository to the Distribution Layer, it this call fails then it will send the next, until the information is stored or all the specified repositories fail.

5.5.4 Service Layer

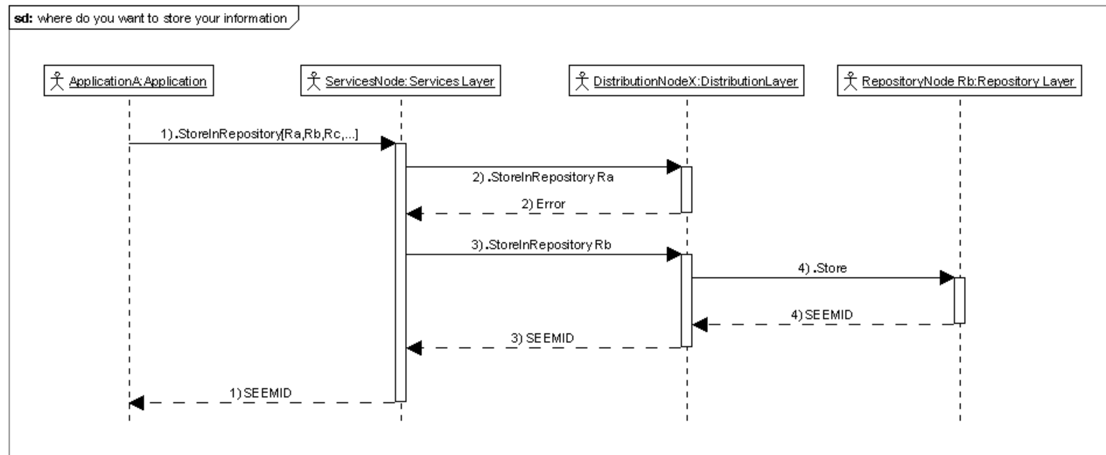
5.5.4.1 Major design decisions

Use the same XLM schema structure as the node baring functionality. If there is some information in the forbidden field then it will be discarded.
All the store information methods will be updated or enhanced to support a new parameter containing the nodeList.

The interfaces to be updated are:
- storeRawDocument
- store
- storeMetadata

5.5.4.2 High level description

A loop of store functions will be sent to the Distribution Layer until the information is stored or the list of allowed nodes has been processed and the information has not been stored. In this case an exception will be thrown. If the information is stored, the SEEMID of the stored data will be returned.



5.5.4.3 Component interface description

See methods to be updated in the final interface description.

5.5.5 Distribution Layer

5.5.5.1 Major design decisions

A new parameter will be included in all the storage functions in order to specify the specific repository where the information has to be stored. Each distribution node has to search for its attached repositories and if it is directly connected to the targeted one then attempt to store the information there. If this operation fails, an exception should be triggered to the Services layer. If the target repository is not attached to the current Distribution node then it has to relay the storage action to its Distribution neighbours in a sequential way.

When an update function is called, the information has to be updated in the same repositories where the original information is.

5.5.5.2 High level description.

Interfaces to be updated:

- storeDocument
- storeMetadata

5.5.5.3 Component interface description

See interfaces to be updated in the final interface description.

5.5.6 Repository Layer

5.5.6.1 Major design decisions

Use the transaction id check in order not to process the same transaction twice.

Project co-funded by the European Commission within the
Sixth Framework Programme, Information Society Technologies (IST)

5.5.6.2 High level description

See transaction Id check in performance chapter.

5.5.6.3 Component interface description

No changes to the repository interface

5.6 Megaping and rating

5.6.1 Functional description

The Megaping functionality returns an XML tree representation of the nodes network, obtaining it in a recursive way, and calculating the time difference between the node call and response.

Some new additional information extraction should be implemented in the different nodes in order to have information about:

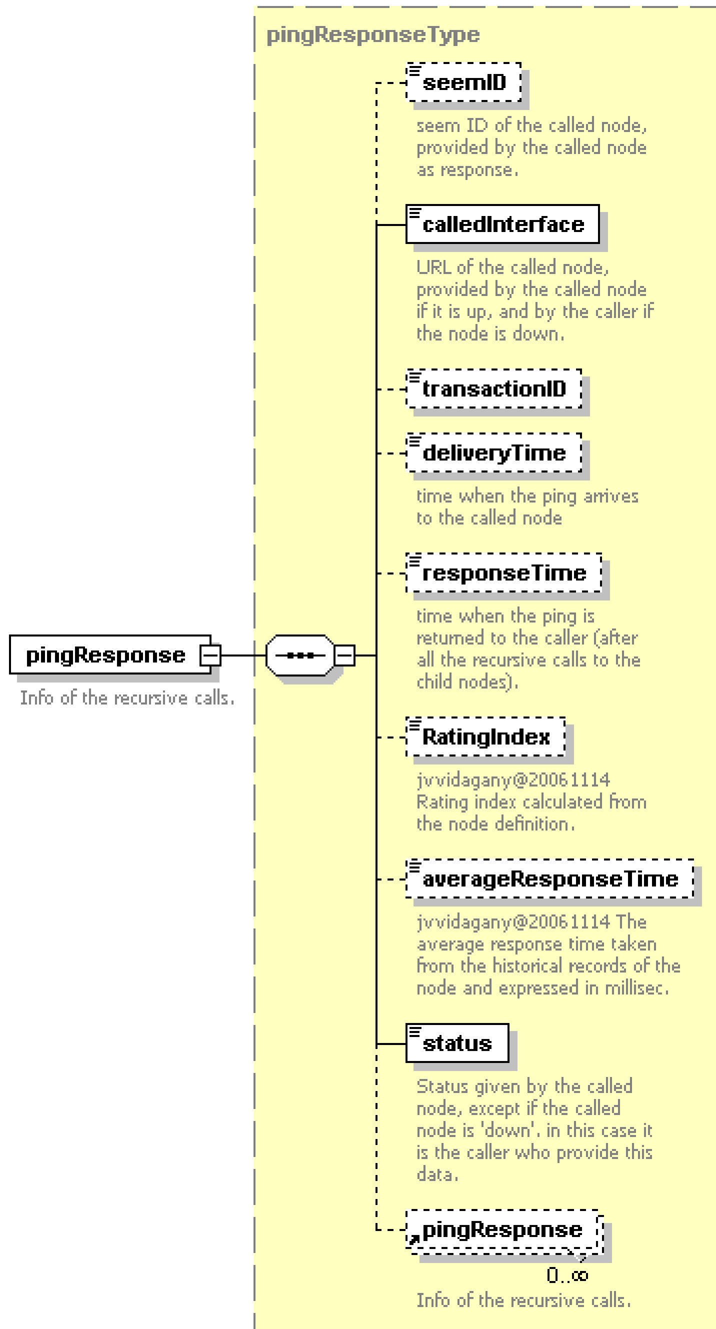
5.6.1.1 Rated nodes based on historical information regarding performance, up time, response times, response times for test queries, etc…

- Node information
- Lists of nodes SEEMIDs.

Based on the historical dates of a specific node, the different nodes that have direct access it can make a rating of it. As in a peer to peer based network a specific node can be accessed from any other node, the routes to access the information (megaping base information) are different. Having that, each node rating functionality will store information on the node definition in order to expose to the rest of the network what is its particular experience with this specific node.

5.6.2 Major design decisions

The response XML schema has to be changed in order to include the new average response time and rating index.



Nodes in the Service and Distribution Layer have to be able to maintain a historical record of the megaping calls made, in order to calculate average times and ratings. Once the node has attended the megaping, the rating entries in the Node description have to be updated in order to have global rating information.
New parameters have to be also included in order to filter the amount of data that has to be returned (include or not the averageResponseTime,etc…).

5.6.3 High level description

Once a megaping call is issued, the node (Service node or Distribution node) has to store the neighbour's nodes information in a historical record and calculate the average and rating following an agreed rating function that will be specified further once this rating has been calculated, the metadata that described this neighbour node will be updated.
So a new node metadata has to be introduced, with the following XML structure:

<ratingInfo>
<seemID="XXXX" rating="6">
<seemID="YYYY" rating="4">
<seemID="ZZZZ" rating="8">
</ratingInfo>

The megaping XML document that results of the execution all over the SRRN, or the subset of nodes specified (see node baring) is build in a recursive way, so the caller node receives a valid pingResponse XML document that has to include in its own pingResponse response to be returned to its caller node.

5.6.4 Service Layer

5.6.4.1 Major design decisions

See 5.5.2.
Interfaces to be updated:
- megaping

5.6.4.2 High level description

See 5.6.3

5.6.4.3 Component interface description

See interfaces to be updated in the final interface description.

5.6.5 Distribution Layer

5.6.5.1 Major design decisions

See 5.5.2.
Interfaces to be updated:
- megaping

5.6.5.2 High level description

See 5.6.3

5.6.5.3 Component interface description

See interfaces to be updated in the final interface description.

5.6.6 Repository Layer

5.6.6.1 Major design decisions

5.6.6.2 High level description

5.6.6.3 Component interface description

5.7 Enhanced notification

5.7.1 Functional description

Notifications allow clients (peers, applications, users) to track information about the objects

lifecycle.
The Service Layer is the only layer involved in the management and notification of changes in the objects lifecycle.
Apart from sending the notifications via email, a new type of notification via Web Services is created and the notifications are now allowed to receive the changes via the documents or metadata when the event, that has occurred, is related to an update of the object content (document or metadata is updated).

5.7.2 Major design decisions

The notification schema document changes to include the seemID of the client that triggers the notification and the notification service (optionally) will attach the difference file between the new version of a document or metadata involved in a notification.

The new notification schema is as follows:

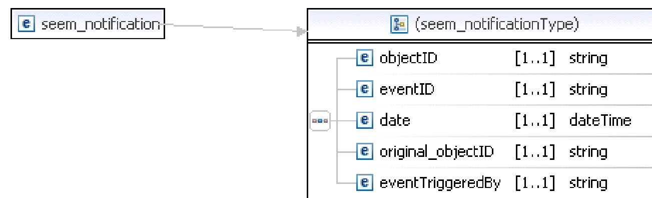


Figure 6 notification schema

A new notification type is added so the notifications can be received in a web service method that act as a notification listener provided at the client side.

The notification information may include a difference file between the last version and the new versions of a document or metadata so that the changes can be analysed and the new version will be downloaded if it is necessary. For example when an ontology changes and a notification is received, if the difference file is attached it is possible to decide if it is necessary to download the new version of the repository.

5.7.3 High level description

To receive notifications via web services the first prerequisite is that the receiver has a notification listener service available to which the Service Layer will send the notifications. The notification listener must implement a web service that complies with the WSDL definition below:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:NotificationListener"
mlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="urn:NotificationListener"
xmlns:intf="urn:NotificationListener"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.2.1
Built on Jun 14, 2005 (09:15:57 EDT)-->

  <wsdl:message name="NotificationListenerResponse">
  </wsdl:message>

  <wsdl:message name="NotificationListenerRequest">
    <wsdl:part name="seemseed_notification_xml" type="soapenc:string"/>
  </wsdl:message>
```

```
  <wsdl:portType name="NotificationListenerInterface">
    <wsdl:operation name="NotificationListener" 
parameterOrder="seemseed_notification_xml">
      <wsdl:input message="impl:NotificationListenerRequest" 
name="NotificationListenerRequest"/>
      <wsdl:output message="impl:NotificationListenerResponse" 
name="NotificationListenerResponse"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="NotificationListenerSoapBinding" 
type="impl:NotificationListenerInterface">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="NotificationListener">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="NotificationListenerRequest">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" 
namespace="urn:NotificationListener" use="encoded"/>
      </wsdl:input>
      <wsdl:output name="NotificationListenerResponse">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" 
namespace="urn:NotificationListener" use="encoded"/>
      </wsdl:output>

    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="NotificationListenerInterfaceService">
    <wsdl:port binding="impl:NotificationListenerSoapBinding" name="NotificationListener">
      <wsdlsoap:address location="http://localhost/services/NotificationListener"/>
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

The parameter *notificationType* in the Service Layer method *subscribeRepositoryObject* and *modifySubscription* now accepts the value “ws” to specify that the notifications will be sent via a web service and the value of the parameter *notificationURI* must be the endpoint url where the notification listener service is deployed. In the notifications, received via web services, the Service Layer will put the same information as in the notifications via email, so an xml with the notification information is received.

A new parameter *receiveDiff* is added to the *subscribeRepositoryObject* and *modifySubscription* methods in Service Layer to receive in the notification a difference file between the last content and the actual content of a document or metadata when an object has been modified. In both notification via email and notification via Webservice, the difference file will be included as an attachment file in the email or in the web service.

The method use by the Service Layer to generate the difference files between metadata and document will be exposed in the Service Layer interface as a utility for the rest of the applications and layers.

5.7.4 Application layer

5.7.4.1 Major design decisions

Applications with software components for receiving and processing object notifications must

refactor its components to handle the notification information in the new xml schema. Also the applications that perform subscriptions or subscriptions modifications need to update the call interfaces for subscribe and modify subscription methods. The applications can of course still receive notifications via email or can choose to receive notifications via web service.

5.7.4.2 High level description

If an application chooses to receive notifications via service web, it has to implement a web service with the WSDL defined in 5.7.3 and deploy it in its application layer so the service is exposed to the Service Layer.
With web services the application doesn't need to check the email account continuously looking for new notifications for processing, the notifications sent by the Service Layer are received directly.

Usually an application processes notifications when it's important to know as soon as possible if any object (document or metadata) has changed its content. For example, in application A the user can approve or disapprove the documents generated in the application B. In this case the users of application A need to know when the documents are created or are modified so they can check them and perform the actions of approval or disapproval. If application B makes subscriptions to every document generated, when a change in the document is made automatically the application A receives the notifications and is well-informed.

The difference file between versions in the notifications listener web service is sent as an attachment and can be retrieved as usual for attachments in web services. Note that the attachment isn't in the WSDL but it must be considered.

5.7.4.3 Component interface description

No changes to the interface

5.7.5 Service Layer

5.7.5.1 Major design decisions

All changes related to notifications enhancement are focused in the Service Layer interface.

5.7.5.2 High level description

The subscribeRepositoryObject method adds a new parameter receiveDiff that sets at subscription time if the notifications will include a difference file between versions. If receiveDiff is true the difference file will be included as attachment else the difference file isn't attached.

The modifySubscription method adds a new parameter receiveDiff that sets if the notifications will include a difference file between versions. If receiveDiff is true the difference file will be included as an attachment else the difference file isn't attached.

The internal procedure getDifferenceBetweenVersions used by the Service Layer is exposed in the Service Layer for the rest of layers. The method gets two documents, metadata or any content as byte arrays and returns the difference file between them.

5.7.5.3 Component interface description

String subscribeRepositoryObject (String *objectID*, String *transactionID*, int *eventId*, String *notificationType*, String *notificationURI*, int *interval*, boolean *receiveDiff*, String *userPublicKey*) throws RemoteException

The method subscribes a user to an object.

Returns:

The return value is an identifier of the subscription or NULL if failed

Parameters:

objectID seemid of the object to subscribe
transactionID Transaction id associated to request
eventId Event identifier to subscribe. Valid values are:
0 - All events
1 - Modify
2 - Delete
3 - Change owner
4 - Change status
5 - Change version
notificationType Type/Channel of notification. Two types of notifications are available: "email" for notifications via email "ws" for notifications via web service. The use of a detailed XML specification is reserved for future versions.
notificationURI The string contains a URI for notification. If the notification type is "email" then the value should be the email address where to send the notification info to, else if the notification type is "ws" then the value should be the url endpoint were the web service notification listener is deployed.
interval Interval of the test of the subscription conditions (in minutes, 0 for default/hourly). *receiveDiff* If true the differences file between the last version and the actual version of a document or metadata is attached when a change or new version event occurs.
userPublicKey Public key of user that invoke the services

Exceptions:

RemoteException Exception in case of an error.

String modifySubscription (String *objectID*, String *transactionID*, int *eventId*, String *notificationType*, String *notificationURI*, int *interval*, boolean *receiveDiff*, String *userPublicKey*) throws Exception

The method modifies the specified subscription with new notification values.

Parameters:

objectID seemid of the object that user is subscribed
transactionID Transaction id associated to request
eventId Event identifier of subscription to modify. Valid values are:
0 - All events
1 - Modify
2 - Delete
3 - Change owner
4 - Change status
5 - Change version
notificationType Type/Channel of notification. Two types of notifications are available: "email" for notifications via email and "ws" for notifications via web service. The use of a detailed XML specification is reserved for future versions.
notificationURI The string contains a URI for notification. If the notification type is "email" then this should be the email address where to send the notification info to, else if the notification type is "ws" then this should be the url endpoint were the web service notification listener is deployed
interval Interval of the test of the subscription conditions (in minutes, 0 for default/hourly). Not implemented yet, instant notification without relay.
receiveDiff If true the differences file between the last version and the actual version of a document or metadata is attached when a change or new version event occurs.
userPublicKey The user's Digital Signature

Exceptions:

RemoteException Exception error

Returns:

The return value is the object identifier to subscribe or NULL if failed

byte [] getDifferenceBetweenVersions (byte *previous_version*[], byte *new_version*[], String *transactionID*) throws Exception

The method returns the difference file between the new version of a document or metadata.

Returns:

The return value is an identifier of the subscription or NULL if failed

Parameters:

previous_version Previous version content
new_version New version content
transactionID Transaction id associated to request

Exceptions:

RemoteException Exception error

5.7.6 Distribution Layer

5.7.6.1 Major design decisions

No implications.

5.7.6.2 High level description

No implications.

5.7.6.3 Component interface description

No implications.

5.7.7 Repository Layer

5.7.7.1 Major design decision

No implications.

5.7.7.2 High level description

No implications.

5.7.7.3 Component interface description

No implications.

5.8 Access information and entry monitoring

5.8.1 Functional description

On the Repository Layer information should be stored on who has accessed the document, how the document was accessed and when the document was accessed.

5.8.2 Major design decisions

- It is recommended that the company information in the Repository Layer will be split in meta data and an XML document. If the meta data is not split the system will be unable to store access log information on company data.
- The access information will be stored in a separate XML database in the Repository Layer.

- The user on the application level can request the access information of documents stored by him self. The SRRN will return an XML document wit detailed access information.
- The access log query is based on the master seem id of the document.
- A user can only request the access log if he is the owner of the document.

5.8.3 High level description

From the application level the user can request access log information from document or company information. Because this information is stored in the Repository Layer new interfaces will be created for all layers of the SRRN.
The information returned from the SRRN is the following XML structure:

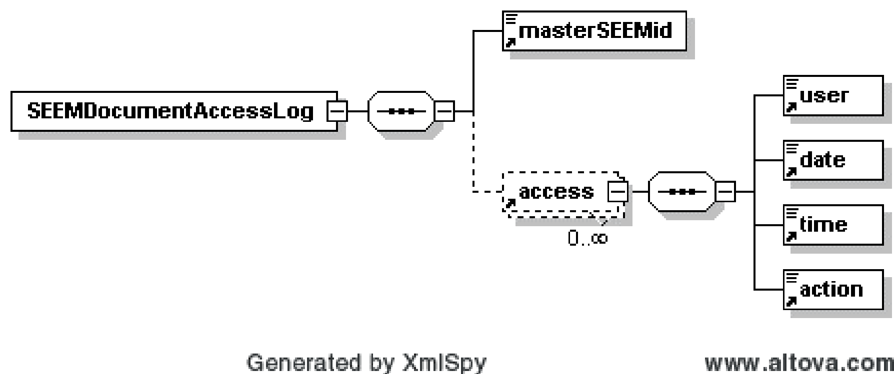


Figure 7: XML schema for document access log

An example XML response file:

```
<SEEMDocumentAccessLog>
	<masterSEEMid>6fa459ea-ee8a-3ca4-894e-db77e160355e</masterSEEMid>
	<access>
		<user>user1</user>
		<date>16112006</date>
		<time>165234</time>
		<action>getdocument</action>
	</access>
	<access>
		<user>user2</user>
		<date>16112006</date>
		<time>194326</time>
		<action>getdocument</action>
	</access>
</SEEMDocumentAccessLog>
```

The document can be stored in more than one repository. Each of the repositories will have his own access information of the document. The result received by the application layer will be the aggregated access information.

5.8.4 Application layer

5.8.4.1 Major design decisions

- A user is only allowed to request an access log from a document if it is the owner of the document.

5.8.4.2 High level description

For the current SRRN all the company information is stored in the metadata and no document is attached to the company information. When a user selects in his application the details for a company then no subsequent call is made to the SRRN because all the information is already available to application. It is recommended for Seamless that for company information an XML document will be attached and the application will make a subsequent call to the SRRN when details for a company are requested by the user.

5.8.5 Service Layer

5.8.5.1 Major design decisions

- The access information interface for the Service Layer is part of the Service Layer Query Services Interface.

5.8.5.2 High level description

The Service Layer should forward the access information query to the connected Distribution Layer and return the result of this query to the application layer.

5.8.5.3 Component interface description

String QueryAccessLog(string transactionId, string userPublickey, string masterseemId, string timestamp)
This method from the query interface performs a query for access information for a document stored on the SRRN and returns an XML document with the access information.

Parameters:
transactionId – The transaction id associated to the query.
userPublickey – The users digital key.
masterseemId – The id for the documents stored on the SRRN.
Timestamp – The timestamp after which the query should be ignored.

Boolean QueryAsyncAccessLogReturnContent(string transactionId, string userPublickey, string masterseemId, string timestamp)
This method performs a query for access information for a document stored on the SRRN and returns an XML document with the access information.

Parameters:
transactionId – The transaction id associated to the query.
userPublickey – The users digital key.
masterseemId – The id for the documents stored on the SRRN.
Timestamp – The timestamp after which the query should be ignored.

5.8.6 Distribution Layer

5.8.6.1 Major design decisions

- The Distribution Layer is responsible for aggregating the results from all connected distribution nodes and repository nodes.

5.8.6.2 High level description

The Distribution Layer broadcasts the access information query to all connected distribution and repository nodes. In the query the master seemid is used because the document could be replicated over multiple repositories. The Distribution Layer will aggregate all incoming results.

5.8.6.3 Component interface description

Bool QueryAsyncAccessLog(string transactionId, string userPublickey, string masterseemId, string timestamp)
This method performs a query for access information for a document stored on the SRRN and returns an XML document with the access information. The query is distributed to all connected Distribution Layers, all connected ebXML/UDDI registry layers and all connected Repository Layers.

Parameters:
transactionId – The transaction id associated to the query.
userPublickey – The users digital key.
masterseemId – The id for the documents stored on the SRRN.
Timestamp – The timestamp after which the query should be ignored.

5.8.7 Repository Layer

5.8.7.1 Major design decisions

- The Repository Layer should only store access information when the document is retrieved and not when meta data is retrieved.
- For the current SRRN the company information is only stored in the metadata and no XML document is attached. It is recommended that the information is split into metadata and an xml document to make it possible to log access to the company information.

5.8.7.2 High level description

The Repository Layer will have a new XML database for storing access information.

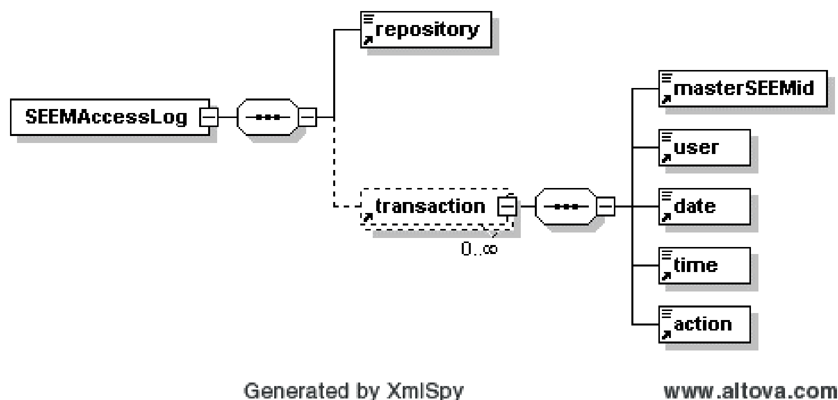


Figure 8: XML schema for access log database

An example XML databbase:

```
<SEEMAccessInformation>
	<repository>a8098c1a-f86e-11da-bd1a-00112444be1e</repository>
	<access>
		<masterSEEMid>6fa459ea-ee8a-3ca4-894e-db77e160355e</masterSEEMid>
		<user>user1</user>
		<date>16112006</date>
		<time>165234</time>
	</access>
```

```
        <access>
                <masterSEEMid>16fd2706-8baf-433b-82eb-8c7fada847da</masterSEEMid>
                <user>user2</user>
                <date>16112006</date>
                <time>231852</time>
        </access>
</SEEMAccessInformation>
```

Based on this database the user can request access information. The user is only allowed to request access information on documents that have been stored by him self (owner). If an access information query is received then the Repository Layer will create an XML document with the detailed information. For a detailed description of this XML document see paragraph 5.8.3

5.8.7.3 Component interface description

String [] QuerySyncAccessLog(string transactionId, string userPublickey, string masterseemId, string timestamp)
This method performs a query for access information for a document stored on the SRRN and returns an XML document with the access information.

Parameters:
transactionId – The transaction id associated to the query.
userPublickey – The users digital key.
masterseemId – The id for the documents stored on the SRRN.
Timestamp – The timestamp after which the query should be ignored.

5.9 Enhanced auditing

5.9.1 Functional description

The enhanced auditing extends the functionality of the existing logging functionality on the SRRN to provide mediator a wide auditing control. The existing logging functionality locally in each layer information about execution information and it exports in an xml defined by an xsd schema so no changes in the existing logging model are necessary.
New simple functionalities are added based on the querying for information about transactions between dates so mediators can collect simple transaction reports.

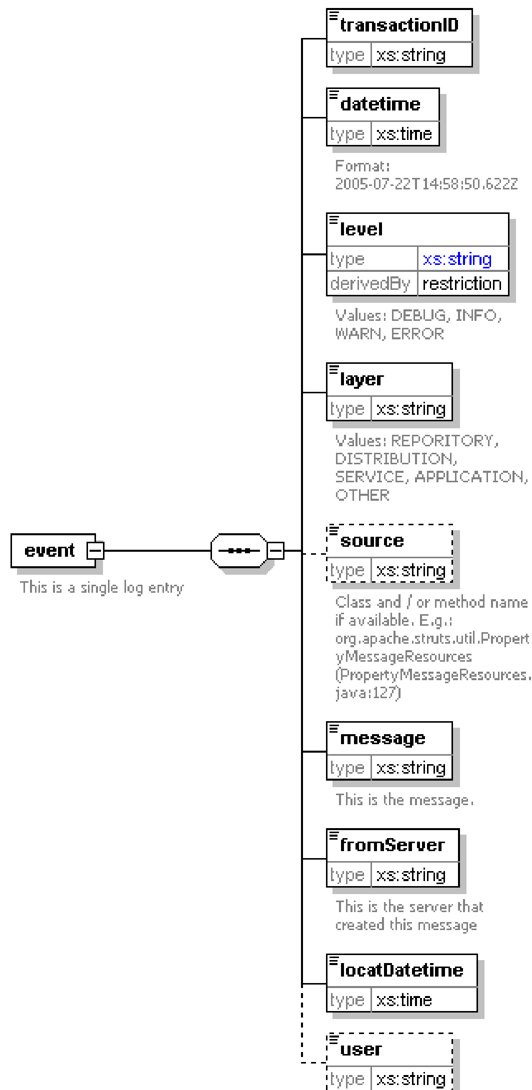
5.9.2 Major design decisions

Even the just stored log transaction records have information about execution time so the internal layer process that store transaction information don't need to be updated but the layer in SRRN must expose service methods for collect that information between dates in all the SRRN layers, so new functions are added in the interfaces:

- Get transaction logs between dates in a layer.
- Get transaction logs between dates recursively in a layer and in its underlying layers.

5.9.3 High level description

Independently the way that each layer stores locally the transaction logging information, the logging information is returned using xml documents that follow the same schema as are used by the logs:



When a get auditing call recursively arrives to a layer, if that layer isn't the last layer in the recursively chain, the layer will pass the recursively call to its immediately lower layer. When the lower layer returns the auditing information, the layer will add its auditing information and the result will be returned to the upper layer.

5.9.4 Application layer

5.9.4.1 Major design decisions

No specific considerations at this layer

5.9.4.2 High level description

No specific considerations at this layer

5.9.4.3 Component interface description

No specific considerations at this layer

5.9.5 Service Layer

5.9.5.1 Major design decisions

Same as in paragraph 5.9.1. No specific considerations at this layer.

5.9.5.2 High level description

The methods getAuditingInfoBetweenDates and getRecursivelyAuditingInfoBetweenDates service methods had to be implemented in the Service Layer interface following the description below. Both methods getAuditingInfoBetweenDates and getRecursivelyAuditingInfoBetweenDates are similar to the existing getLogs and getLogsRecursively in the Service Layer but instead of getting the execution information of only a transaction they return the execution information of all transactions that have occurred between two dates.

In the getAuditingInfoBetweenDates method if the layer specified in the parameter *layer* is an underlying layer then the execution is delegated to the getAuditingInfoBetweenDates method in the next lower layer and so on until it arrives to the layer specified in the parameter *layer.* After collecting the auditing info between dates in the requested layer the results are uploaded through the layers until they reach the application layer.

When a getAuditingInfoBetweenDates method request arrives to layer A:

1. If the layer specified in the parameter *layer* is a lower layer then layer A in the SRRN hierarchy then the execution is delegated to the getAuditingInfoBetweenDates method in the next lower layer in the hierarchy. This step is repeated until the layer specified in the parameter layer is reached.
2. In the requested layer the auditing information is collected and is send to the next upper layer until it reaches the application layer.

When a getRecursivelyAuditingInfoBetweenDates method request arrives to layer A:

1. If in the specified layers in parameter *layer* there is a layer under the layer A in the SRRN hierarchy then a request for getRecursivelyAuditingInfoBetweenDates with the same parameters to the next lower layer in the hierarchy is made. This step is repeated until the more inner layer in the recursive request is reached.
2. From the most inner layer in the recursively requests the auditing information is added and is send to the next upper layer in the hierarchy. This step is repeated until the results reached the application layer.

5.9.5.3 Component interface description

String getAuditingInfoBetweenDates (String *info_level*, int *layer*, Date *from*, Date *to*, String *userPublicKey*) throws Exception
Method to retrieve the transaction information log generated in a specific layer between dates
Returns:
This method returns an xml nodelist of log entries as defined above.
Parameters:
from Date from to retrieve transaction information logs
to Date up to retrieve transaction information logs
info_level Info level of logs to retrieve: DEBUG, INFO, WARN or ERROR
layer Layer from retrieve information logs. Values:
0: Application Layer
1: Service
2: Distribution
3: Repository
userPublicKey Transaction id associated to request
Exceptions:

Exception Throws an exception in case of an error

String getRecursivelyAuditingInfoBetweenDates (String *info_level*, int *layer*, Date *from*, Date *to*, String *userPublicKey*) throws Exception

Method to retrieve the transaction information log generated in a specific layer between dates and its underlying layers

Parameters:

from Date from to retrieve transaction information logs
to Date up to retrieve transaction information logs
info_level Info level of logs to retrieve: DEBUG, INFO, WARN or ERROR
layer Layer from retrieve information logs. Values:
0: Application Layer
1: Application, Service
2: Application, Service, Distribution
3: Application, Service, Distribution, Repository
userPublicKey Transaction id associated to request

Returns:

This method returns a xml nodelist of log entries as defined above.

Exceptions:

Exception Throws an exception in case of an error

5.9.6 Distribution Layer

5.9.6.1 Major design decisions

The methods getAuditingInfoBetweenDates and getRecursivelyAuditingInfoBetweenDates service methods had to be implemented in the Distribution Layer interface following the description below.

5.9.6.2 High level description

Same as in 5.9.5.2 . No specific considerations at this layer.

5.9.6.3 Component interface description

Same as in 5.9.5.3 . No specific considerations at this layer.

5.9.7 Repository Layer

5.9.7.1 Major design decisions

The methods getAuditingInfoBetweenDates and getRecursivelyAuditingInfoBetweenDates service methods had to be implemented in the Distribution Layer interface following the description below.

5.9.7.2 High level description

Same as in 5.9.5.2 . No specific considerations at this layer.

5.9.7.3 Component interface description

Same as in 5.9.5.3 . No specific considerations at this layer.

5.10 Security

Since security is crucial for the success of SEAMLESS, a comprehensive security concept is used within the whole project that goes beyond the distributed storage system. This concept is described in the Security Deliverable and it includes the functional specification of the distributed storage system in the security domain.

5.11 Content retrieval

5.11.1 Functional description

The SRRN was not developed with performance and ‘real-use’ in mind since it was a policy oriented project.
In order to handle the number of mediators and users from SEAMLESS, the architecture of the SRRN should be thoroughly examined to improve the performance.

5.11.2 Major design decisions

- The Service Layer determines which aggregation method is used, aggregation on the Service Layer or aggregation on the Distribution Layer.

5.11.3 High level description

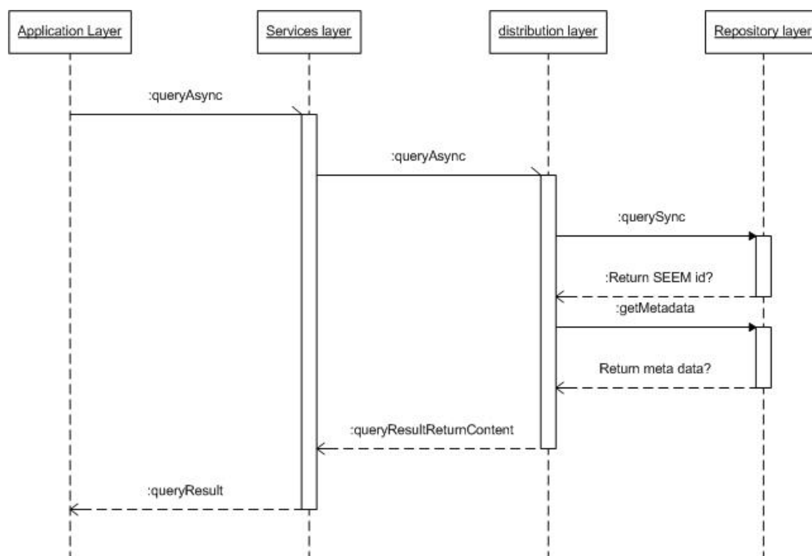


Figure 9: Aggregation

5.11.4 Service Layer

5.11.4.1 Major design decisions

- The Service Layer will have a configuration setting for the aggregation method used. Aggregation on the Service Layer or aggregation on the Distribution Layer.

5.11.4.2 High level description

The Service Layer will submit the query to the Distribution Layer. If aggregation on distribution

level is used the Service Layer gets the metadata back instead of the seemid's and the Distribution Layer will not be queried for meta data.

5.11.4.3 Component interface description

5.11.5 Distribution Layer

5.11.5.1 Major design decisions

5.11.5.2 High level description

If the aggregation is performed on Distribution Layer level then the results of a query to a repository node are not returned to the services layer or the higher distribution node but the Distribution Layer should immediately request the metadata of the seemid and return the metadata to the upper layer.

5.11.5.3 Component interface description

The QueryAsync interface will have an extra boolean parameter specifying the aggregation level:
Bool UseDistributionAggregation
In case this boolean is true then the Distribution Layer is responsible for requesting the meta data after the seemid's are received.

6 SRRN layers interfaces description

6.1 Services Layer

6.1.1 Object Lyfecycle Services Interface

String store (byte[] *document*, String *metadataRDF*, String *version*, String *newAccessRights*, String *transactionID*, String *userPublicKey*) throws Exception

The method stores a document in the repository.

Parameters:

version Initial document version. If this parameter is null then the version information isn't added to document.
newAccessRights xml with the access rights to be assigned to seem object
transactionID Transaction id associated to request
userPublicKey The user's Digital Signature.
document document (BASE64) to be stored in the repository. If the encoding of the attached document is DIME or MIME, then the document is included in the method call, so the document parameter will be null. The BASE64 attachment has been included to facilitate mobility stuff.
metadataRDF Metadata describing the document. If null, then a basic metadata is associated to the document.

Returns:

Unique identifier of the stored document.

Exceptions:

Exception Throws a lifeCycleError exception in case of an error

string storeRawDocument(Object document)

The method stores a document in the repository. During storage metadata for the document is generated by the service. The metadata includes only the identity of the creator, creation date.

Parameters:

document - the document to be stored

Returns:

The unique identifier of the stored document

Exceptions:

Throws a LifeCycle exception in case of an error

String storeMetadata (String *metadata*, String *version*, String *newAccessRights*, String *transactionID*, String *userPublicKey*) throws Exception

This method stores metadata entries in the repository following the RDF format defined for storage of metadata at the repositories. When creating the new entry, the repository generates a unique identifier for it.

Returns:

The identifier of the stored metadata entry of the stored metadata if succeed.

Parameters:

version Version of the document. If version is null then no version is added to document. If version is "AUTO" then is generated automatically.
newAccessRights xml with the access rights to be assigned to seem object
transactionID Transaction id asociated to request
userPublicKey The user's Digital Signature.
metadata a string containing the metadata that should be stored, in RDF format

Exceptions:

Exception Throws a LifeCycle exception in case of an error.

bool updateMetadata (String seemID, String metadata)

Updates the metadata entry specified by the provided identifier.

Parameters:

seemID - the identifier of the desired metadata entry
metadata - a string containing the new metadata that should be stored, in RDF format

Returns:

True if the update was successful, False otherwise (seemID does not exist).

Exceptions:

Throws a LifeCycle exception in case of an error.

bool updateDocument (String seemID, Object document)

The method updates an existing document. The user has to provide the unique identifier of the document and the document itself.

Parameters:

seemID - the identifier of the desired document
document - the document to be updated

Returns:

True if the update was successful, False otherwise

Exceptions:

Throws a LifeCycle exception in case of an error.

bool remove (string seemID)

Removes the entry document and/or metadata specified by the provided identifier. In case of a document, the method deletes *both* document and metadata.

Parameters:

seemID - the identifier of the desired entry

Returns:

True if the removal was successful; False otherwise (seemID does not exist)

Exceptions:

Throws a LifeCycle exception in case of an error.

boolean addPreDefinedNodeList (String listName, String description, String NodeList) throws Exception

Method that allow to add new named node lists for implementing node baring functionality

Parameters:

listName name of the node list, to be retrieved afterwards. Univocally identifies each node list.
description textual description of the list
NodeList List of nodes following the nodeList schema.

Returns:

true in case the operation is successfully executed.

Exceptions:

Exception If any error occurs, then an exception is thrown.

Boolean deletePreDefinedNodeList (String listName) throws Exception

Node baring - Method that allow to delete a named node lists.

Parameters:

listName name of the node list, to be retrieved afterwards. Univocally identifies each node list.

Returns:

true in case the operation is successfully executed.

Exceptions:

java.rmi.RemoteException If any error occurs, then an exception is thrown.

Boolean modifyPreDefinedNodeList (String listName, String description, String NodeList) throws Exception

Node Baring - Method that allow modifying a node list

Parameters:

listName name of the node list to be retrieved modified.
description modified textual description of the list
NodeList Modified list of nodes following the nodeList schema.

Returns:

true in case the operation is successfully executed.

Exceptions:

java.rmi.RemoteException If any error occurs, then an exception is thrown.

6.1.2 Query Services Inteface

String [] queryMetadata (String *transactionID*, String *userPublicKey*, String *srrnQuery*, String *xmlQueryDescriptor*) throws Exception

The srrnQuery allows the user to search for metadata by using both the XML query

format following the defined XSD for XML query, or SPARQLquery.

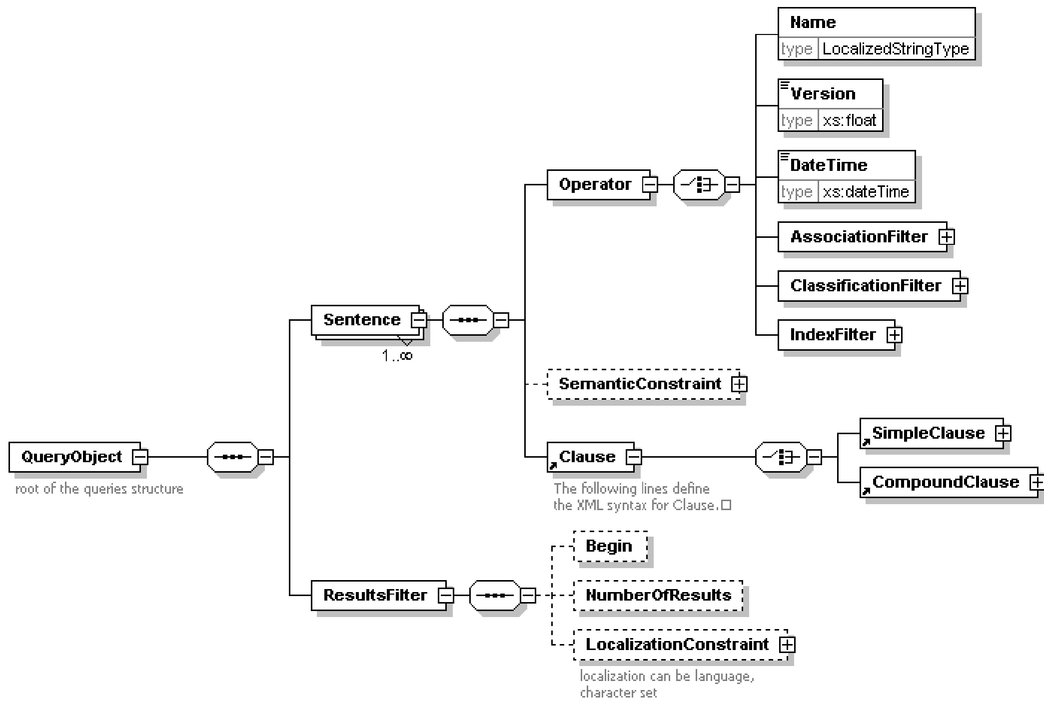


Figure 10 Query object

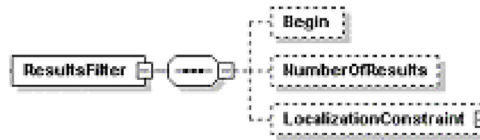


Figure 11 Query descriptor

Now SPARQLqueries return metadata or attribute values from metadata. If there isn't attribute values in the SPARQLselect clause then return the seem_id of metadata that matches criteria else if there is attribute values in the SPARQLselect clause return an xml document with the attribute values from the metadata that matches criteria.

Example:

select ?name, ?address where (?user,<kn:name>,?CONSTRAINTO), ….. and ?CONSTRAINT0 eq …. USING kn for <http://www.seemseed.net/metadata/seem#>

Result:

```
<result>
<name>name1</name>
<address>address1</address>
</result>
<result>
<name>name2</name>
<address>address2</address>
</result>
....
<result>
```

<name>name_n</name>
<address>address_n</address>
</result>

Parameters:

transactionID Transaction id associated to request
userPublicKey The user's Digital Signature
srrnQuery The method expects a string containing an SPARQLquery, or and XML query as defined in the previous section.
xmlQueryDescriptor The queryDescriptor allow to filter the amount and characteristics of the results, it allow to define how many results are expected, the time-out for the query, some localization features related to the results,...

Returns:

The return value of the query is an array of IDs of objects that matches the query criteria or an xml document which structure is defined as a node list with the attributes of the select clause.

Exceptions:

Exception Throws an queryError exception in case of an error

String [] queryMetadataEx (String *transactionID*, String *userPublicKey*, String *srrnQuery*, String *xmlQueryDescriptor, string nodelist*) throws Exception

The srrnQuery allows the user to search for metadata by using both the XML query format following the defined XSD for XML query, or SPARQLquery.

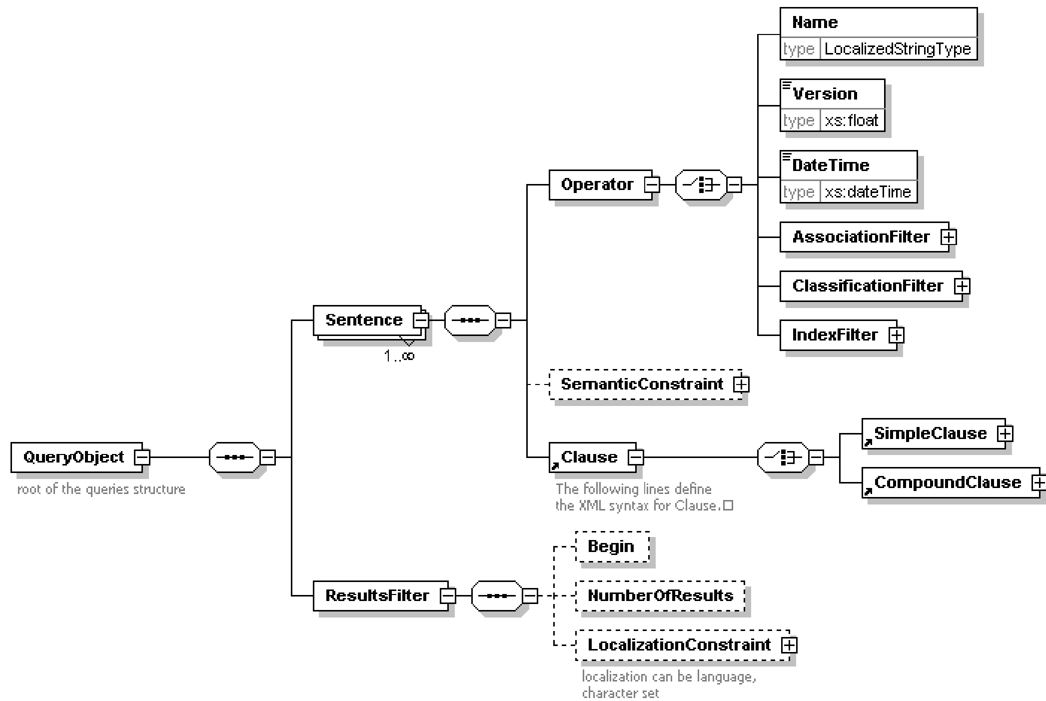


Figure 12 Query object

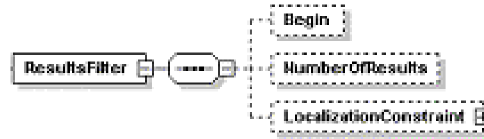


Figure 13 Query descriptor

Now SPARQLqueries return metadata or attribute values from metadata. If there isn't attribute values in the SPARQLselect clause then return the seem_id of metadata that matches criteria else if there is attribute values in the SPARQLselect clause return an xml document with the attribute values from the metadata that matches criteria.

Example:

select ?name, ?address where (?user,<kn:name>,?CONSTRAINTO), ….. and ?CONSTRAINT0 eq …. USING kn for <http://www.seemseed.net/metadata/seem#>

Result:

```
 <result>
<name>name1</name>
<address>address1</address>
</result>
<result>
<name>name2</name>
<address>address2</address>
</result>
....
<result>
<name>name_n</name>
<address>address_n</address>
</result>
```

Parameters:

transactionID Transaction id associated to request
userPublicKey The user's Digital Signature
srrnQuery The method expects a string containing an SPARQLquery, or and XML query as defined in the previous section.
xmlQueryDescriptor The queryDescriptor allow to filter the amount and characteristics of the results, it allow to define how many results are expected, the time-out for the query, some localization features related to the results,...
nodelist xml structure fefining the nodes that are allowed and not allowed to use for the query.

Returns:

The return value of the query is an array of IDs of objects that matches the query criteria or an xml document wich structure is defined as a node list with the attributes of the select clause.

Exceptions:

Exception Throws an queryError exception in case of an error

string getMetadata (string seemID)

The method retrieves a single metadata entry based on its seemID. The metadata is returned in RDF format.

Parameters:

seemID - the seem identifier of the requested metadata entry

Returns:

An RDF string containing the retrieved metadata entry, or an empty string if no metadata entry with the specified identifier could be found in the storage or access violates security rules.

Exceptions:

exception - Throws a queryError exception in case of an error

string getMetadataEx (string seemID, string nodelist)

The method retrieves a single metadata entry based on its seemID. The metadata is returned in RDF format.

Parameters:

seemID - the seem identifier of the requested metadata entry
nodelist xml structure fefining the nodes that are allowed and not allowed to use for the query.

Returns:

An RDF string containing the retrieved metadata entry, or an empty string if no metadata entry with the specified identifier could be found in the storage or access violates security rules.

Exceptions:

exception - Throws a queryError exception in case of an error

Object getObject (string seemID)

Retrieves an object based on its seemID. The method returns the object. In case the object does not exist, the return value is empty.

Parameters:

seemID - the identifier of the requested object

Returns:

The requested object or empty if object does not exist or access violates security rules.

Exceptions:

exception - Throws a queryError exception in case of an error

String ping ()

The methods returns information on the operator of the service and the terms of use.

Parameters:

n/a

Returns:

Information on the operator of the service and the terms of use.

Exceptions:

n/a

String megaping (String transactionID)

Method that allows to know the status of all nodes.

Parameters:

transactionID – Transaction id associated to request.

Returns:

Returns an xml with the state information of the nodes. The return information is a concatenation of xml pingResponse node using a specific XML schema.

Exceptions:

Throws a queryError exc

The following internal interfaces are defined:

Boolean validateQuerySyntax (String srrnQuery)

This internal method provides a syntax check for SPARQLqueries. The check is provided to catch all potential security risks at this early stage of processing.

Parameters:

srrnQuery – string containing the query to be validated

Returns:

Boolean, true if successful, false otherwise.

String filterQuery (Object response, String xmlQueryDescriptor)

The filter method filters all returned objects form the Core registry according to the query descriptor.

boolean queryAsync (String *query*, long timestamp, String *fromEndpointURL*, String *fromServiceURN*, String *transactionID*, String *userPublicKey*) throws Exception

This method performs an asynchronous search on the metadata objects stored in the registry. When this method is executed it translates the query to registry and returns the control immediately. As soon as the registry is going collect query results, the results are returned calling the queryResult or queryResultContent methods in the fromEndpointURL and fromServiceURN provided by the application layer. Works as the queryMetadata method but return results in a asynchronous way.

Parameters:

query The RDF query to execute.
timestamp Timestamp after which incoming results should be ignored.
fromEndpointURL Endpoint from application layer where receive results
fromServiceURN Service URN from application layer where receive results
transactionID Transaction id associated to request
userPublicKey Public key of the user that execute service

Returns:

True if the request is delivered to Distribution Layer, false otherwise.

Exceptions:

Exception Throws an exception in case of an error.

boolean queryAsyncEx (String *query*, long timestamp, String *fromEndpointURL*,

String *fromServiceURN*, String *transactionID*, String *userPublicKey, string nodelist*) throws Exception

This method performs an asynchronous search on the metadata objects stored in the registry. When this method is executed it translates the query to registry and returns the control immediately. As soon as the registry is going collect query results, the results are returned calling the queryResult or queryResultContent methods in the fromEndpointURL and fromServiceURN provided by the application layer. Works as the queryMetadata method but return results in a asynchronous way.

Parameters:

query The RDF query to execute.
timestamp Timestamp after which incoming results should be ignored.
fromEndpointURL Endpoint from application layer where receive results
fromServiceURN Service URN from application layer where receive results
transactionID Transaction id associated to request
userPublicKey Public key of the user that execute service
nodelist xml structure defining the nodes that are allowed and not allowed to use for the query.

Returns:

True if the request is delivered to Distribution Layer, false otherwise.

Exceptions:

Exception Throws an exception in case of an error.

byte [] getDocumentBySeemID (String transactionID, String seemID, String userPublicKey, String type) throws Exception

Method to retrieve documents from the repositories by the metadata seemID that contains the document in the repository.

Parameters:

transactionID - Transaction id associated to request
userPublicKey - Digital Signature of the user that executes the service
seemID - The identifier of the document to be retrieved
type - Type of codification in which the document will be returned. The allowed values are: MIME, DIME and BASE64 (for attachments of less than 1Mb. mainly to work from mobile devices).

Returns:

If the type equals BASE64 then the document is returned as byte array, if not, the document is included into the response following the DIME or MIME codification.

Exceptions:

Exception Throws a queryError exception in case of an error

byte [] getDocument (String transactionID, String documentID, String userPublicKey, String type) throws Exception

Method to retrieve a document from the repositories by its document_ID

Returns:

If the type equals BASE64 then the document is returned as byte array, if not, the document is included into the response following the DIME or MIME codification.

Parameters:

documentID - The document_id of the document to retrieve
transactionID - Transaction id asociated to request
userPublicKey - Digital Signature of the user that executes the service
type - Type of codification in which the document will be returned. The allowed values are: MIME, DIME and BASE64 (for attachments of less than 1Mb. mainly to work from mobile devices).

Exceptions:
Exception Throws a queryError exception in case of an error

String getMetadataIDFromDocument (String *transactionID*, String *documentID*, String *userPublicKey*) throws Exception
Method to retrieve the identifier of the metadata that envelops a document.

Returns:
The seem_id of the metadata that envelops the document in the repository

Parameters:
documentID The document_id of the document to retrieve
transactionID Transaction id asociated to request
userPublicKey Digital Signature of the user that executes the service

Exceptions:
Exception Throws a queryError exception in case of an error

boolean removeAllDocumentVersions (String *transactionID*, String *documentID*, String *userPublicKey*) throws Exception
Removes all the existing versions of the document specified by document_id in the repository. All the metadata that envelop the deleted documents in the repository are deleted too.

Parameters:
transactionID Transaction id asociated to request
userPublicKey The user's Digital Signature.
documentID Id of the object in the repository to be deleted

Returns:
true if success or false if error.

Exceptions:
Exception Throws a LifeCycle exception in case of an error

boolean removeAllMetadataVersions (String *transactionID*, String *seemID*, String *userPublicKey*) throws Exception
Removes all the existing versions of the metadata specified by seem_id in the repository. If the metadata deleted have documents associated, the documents are deleted too.

Returns:
true if success or false if error.

Parameters:
transactionID Transaction id asociated to request
userPublicKey The user's Digital Signature.

seemID Id of the object in the repository to be deleted

Exceptions:
Exception Throws a LifeCycle exception in case of an error

String getLastVersionDocument (String *transactionID*, String *documentID*, String *userPublicKey*) throws Exception

Get the last version of the document specified by document_id in the repository.

Returns:
The document_id of the last version of the document which identifier is documentID.

Parameters:
transactionID Transaction id associated to request
userPublicKey The user's Digital Signature.
documentID Document id from which retrieve all its versions

Exceptions:
Exception Throws a LifeCycle exception in case of an error

String getLastVersionMetadata (String *transactionID*, String *seemID*, String *userPublicKey*) throws Exception

Get the last version of the metadata specified by seem_id in the repository.

Returns:
The seem_id of the last version of the metadata which identifier is seemID.

Parameters:
transactionID Transaction id associated to request
userPublicKey The user's Digital Signature.
seemID Metadata seem id from which retrieve all its versions

Exceptions:
Exception Throws a LifeCycle exception in case of an error

String [] getAllDocumentVersions (String *transactionID*, String *document_id*, String *userPublicKey*) throws Exception

Get the document id's from all the existing versions of the document which documentID is equal to document_id in the repository.

Returns:
The document_id's of the existing versions.

Parameters:
document_id The document identifier from which retrieve all the available versions
transactionID Transaction id associated to request
userPublicKey The user's Digital Signature.

Exceptions:
Exception Throws a LifeCycle exception in case of an error

String [] getAllMetadataVersions (String *transactionID*, String *seem_id*, String

***userPublicKey*) throws Exception**

Get the seem id's from all the existing versions of the metadata which seemID is equal to seem_id in the repository.

Returns:

The seem_id's of the existing versions.

Parameters:

seem_id The metadata seem_id from which retrieve all versions available
transactionID Transaction id asociated to request
userPublicKey The user's Digital Signature.

Exceptions:

Exception Throws a LifeCycle exception in case of an error

string [] queryTest (String xmlQueryTest)

The method performs a QueryTest through all the layers and returns an XML structure with the test results.

Parameters:

string xmlQueryTest – This function expects an XML structure with the current test results.

Return value:

string[] - The return value of the query is an XML structure with the results of the test query.

Exceptions:

exception - Throws a queryError exception in case of an error

String [] QueryAccessLog(string transactionId, string userPublickey, string masterseemId, string timestamp)

This method from the query interface performs a query for access information for a document stored on the SRRN and returns an XML document with the access information.

Parameters:

transactionId – The transaction id associated to the query.
userPublickey – The users digital key.
masterseemId – The id for the documents stored on the SRRN.
Timestamp – The timestamp after which the query should be ignored.

Boolean [] QueryAsyncAccessLogReturnContent(string transactionId, string userPublickey, string masterseemId, string timestamp)

This method performs a query for access information for a document stored on the SRRN and returns an XML document with the access information.

Parameters:

transactionId – The transaction id associated to the query.
userPublickey – The users digital key.
masterseemId – The id for the documents stored on the SRRN.
Timestamp – The timestamp after which the query should be ignored.

String getAuditingInfoBetweenDates (String *info_level*, int *layer*, Date *from*, Date *to*, String *userPublicKey*) throws Exception

Method to retrieve the transaction information log generated in a specific layer between dates

Returns:

This method returns an xml nodelist of log entries as defined above.

Parameters:

from Date from to retrieve transaction information logs
to Date up to retrieve transaction information logs
info_level Info level of logs to retrieve: DEBUG, INFO, WARN or ERROR
layer Layer from retrieve information logs. Values:
0: Application Layer
1: Service
2: Distribution
3: Repository
userPublicKey Transaction id associated to request

Exceptions:

Exception Throws an exception in case of an error

String getRecursivelyAuditingInfoBetweenDates (String *info_level*, int *layer*, Date *from*, Date *to*, String *userPublicKey*) throws Exception

Method to retrieve the transaction information log generated in a specific layer between dates and its underlying layers

Parameters:

from Date from to retrieve transaction information logs
to Date up to retrieve transaction information logs
info_level Info level of logs to retrieve: DEBUG, INFO, WARN or ERROR
layer Layer from retrieve information logs. Values:
0: Application Layer
1: Application, Service
2: Application, Service, Distribution
3: Application, Service, Distribution, Repository
userPublicKey Transaction id associated to request

Returns:

This method returns a xml nodelist of log entries as defined above.

Exceptions:

Exception Throws an exception in case of an error

String [] getPreDefinedNodeList ( )

Function that allows retrieving the names of the stored predefined Nodes Lists.

Returns:

List of name,description pairs of the predefined nodeLists stored.

6.1.3 Subscription Services Interface

String subscribeRepositoryObject (String seemID, Integer eventId, String notificationType, String notificationURI, Integer interval)

The method subscribes a user to an object.

Parameters:

String seemID – Identifier of the object to be monitored by the service metadata.
Integer eventId – Event identification, which causes the notification, 0 if all events should be monitored.
String notificationType – Type/Channel of notification e.g. e-mail. For this specification only the string "email" is valid. The use of a detailed XML specification is reserved for future versions.
String notificationURI – The string contains a URI for notification. For this version the URI is limited to an e-mail adress.
Integer interval – Interval of the test of the subscription conditions (in minutes, 0 for default/hourly)

Return value:

String - The return value is an identifier of the subscription, NULL if failed

Exceptions:

exception - Throws a subscriptionError exception in case of an error

String unsubscribeRepositoryObject (String subscriptionId)

The method unsubscribes a user to an object by deleting the subscription with the mentioned id.

Parameters:

String subscriptionId – Identifier of the subscription to be deleted.

Return value:

Integer - The return value is an error code, 0 if successful

Exceptions:

exception - Throws a subscriptionError exception in case of an error

String listSubscriptions ()

The method returns a list of all subscriptions of this user.

Parameters:

-

Return value:

String – XML string with descriptions of all subscriptions for this user.

Exceptions:

exception - Throws an subscriptionError exception in case of an error

String detailSubscription (String subscriptionId)

The method returns the details of a specific subscription.

Parameters:

String subscriptionId – Identifier of the subscription for which the details should be returned.

Return value:

String – XML string with descriptions of the defined subscription.

Exceptions:

exception - Throws a subscriptionError exception in case of an error

String modifySubscription (String subscriptionId, String seemID, Integer eventId,

String notificationType, String notificationURI, Integer interval)

The method modifies the specified subscription with the values stated in the call. If a value should not be changed it must be set to NULL.

Parameters:

String subscriptionId – Identifier of the subscription which should be altered.
String seemID – Identifier of the object to be monitored by the service
Integer eventId – Event identification, which causes the notification, 0 if all events should be monitored.
String notificationType – Type/Channel of notification e.g. e-mail. For this specification only the string "email" is valid. The use of a detailed XML specification is reserved for future versions.
String notificationURI – The string contains a URI for notification. For this version the URI is limited to an e-mail address.
Integer interval – Interval of the test of the subscription conditions (in minutes, 0 for default/hourly)

Return value:

String - The return value is an identifier of the subscription, NULL if failed

Exceptions:

exception - Throws a subscriptionError exception in case of an error

6.1.4 Application Service Interface

String deployApplicationServices (byte *service_code_jar_binary*[], String *service_wsdl*, String *transactionID*) throws Exception

Deploy a pluggable application service extension for share with other applications.

Returns:

The service endpoint url in the Service Layer

Parameters:

transactionID Transaction identifier of the request
service_code_jar_binary Java binary .jar file with the service implementation
service_wsdl wsdl of the service to deploy

Exceptions:

Exception Error if fails the deployment

String modifyApplicationServices (byte *service_code_jar_binary*[], String *service_wsdl*, String *transactionID*) throws Exception

Modify an existing pluggable application service extension for share with other applications.

Returns:

The service endpoint url in the Service Layer

Parameters:

transactionID Transaction identifier of the request
service_code_jar_binary Java binary .jar file with the modified service implementation
service_wsdl wsdl of the modified service

Exceptions:

Exception Error if fails the deployment

boolean deleteApplicationServices (String *pluggable_service_endpoint_url*, String *transactionID*) throws Exception

Remove an existing pluggable application service extension.

Returns:

true if the service is undeployed or false if error

Parameters:

transactionID Transaction identifier of the request
pluggable_service_endpoint_url Service Endpoint of the pluggable service to remove.

Exceptions:

Exception Error if fails the undeploy

6.2 Distribution Layer

6.2.1 Controller component

The Controller component provides methods, listed in 6.2.6

Furthermore, it will provide some internal methods for communication with the other components. A list of these additional methods is defined in the corresponding API documentation.

6.2.2 Repository-connector component

boolean canStoreBinary () throws Exception

This method checks if the data source can be used to store a binary file.
If it succeeds, then the Registry Layer will call the storeDocument method. This includes a decision for updates.

Returns:

Returns true if the data source is able and willing to store the data

Exceptions:

Exception Throws an exception in cases of an error.

boolean canStoreBinaryByName (String *documentName*) throws Exception

This method checks if the data source can be used to store a binary file.
This decision has to be made by the documentName method. If it succeeds, then the Registry Layer will call the storeDocument method. This includes a decision for updates.

Parameters:

documentName the name of the document.

Returns:

Returns true if the data source is able and willing to store the data

Exceptions:

Exception Throws an exception in cases of an error.

boolean canStoreMetadata (String *Metadata*) throws Exception

This method checks if the Metadata file can be stored and later be queried.
If it succeeds, then the Core Registry Layer will probably call the storeMetadata method.

This includes a decision for updates.

Returns:
Returns true if the data source is able and willing to store the metadata.

Exceptions:
Exception Throws an exception in cases of an error.

DataHandler getDocument (String *seemID*) throws Exception

This method can be used to retrieve the whole document, which is described by a metadata file.

Parameters:
seemID The ID of a binary file

Returns:
the file (e.g. an PDF-document).

Exceptions:
Exception Throws an exception in cases of an error.

String getMetadata (String *seemID*) throws Exception

This method can be used to retrieve a metadata based on the ID.

Parameters:
seemID The ID of a metadata/RDF file

Returns:
The metadata. If no metadata exists then null will be returned

Exceptions:
Exception Throws an exception in cases of an error.

String ping () throws Exception

This sends the IP address of the requester.
This method can be used to check, if this webservice is working.

Returns:
The IP address of the remote host

Exceptions:
Exception Throws an exception in cases of an error.

String megaping (String *transactionID*) throws Exception

This method starts a recursive Megaping.

Parameters:

transactionID

Returns:

Megaping xml docuent

Project co-funded by the European Commission within the
Sixth Framework Programme, Information Society Technologies (IST)

Exceptions:

Exception Throws an exception in cases of an error.

boolean removeEntry (String *seemID*) throws Exception

Removes an Entry from the registry/repository. This can be used to remove documents or metadata.

Parameters:

seemID the id of the Entry to be removed

Returns:

returns true if the entry was removed successfully, else false.

Exceptions:

Exception Throws an exception in cases of an error.

String storeDocument (DataHandler *document*) throws Exception

Stores a document and returns an ID.

Parameters:

document the file that should be stored (e.g. an PDF file,...)

Returns:

returns an ID to access the document. ID is unique within the whole SRRN network and is indeed globally unique

Exceptions:

Exception Throws an exception in cases of an error or if the source was not able/allowed to store the file

String storeMetadata (String *metadata*) throws Exception

Stores an RDF String (e.g. the Metadata).
This is useful to store RDF files / Metadata with no documents attached. Those RDF-Strings will only need to have 2 or 3 defined SEEMseed properties (as explained in) and have no other limitations. They can be used to store e.g. classifications and they can be queried directly. The difference to the storeDocument method is that RDF must be searchable via RDQL

Parameters:

metadata an RDF/Metadata to store. This can have additional information and will be searchable

Returns:

returns an ID to access the metadata. ID is unique within the whole network

Exceptions:

Exception Throws an exception in cases of an error or if the source was not able/allowed to store the file

boolean updateDoc (String *seemID*, DataHandler *document*) throws Exception

Updates a document.

Parameters:
seemID the id of the document
document an updated document (e.g. an PDF file,...)

Returns:
Returns true if the update was successful, else false.

Exceptions:
Exception Throws an exception in cases of an error.

boolean updateMetadata (String *seemID*, String *metadata*) throws Exception
Updates an RDF file/metadata. This method will a) catch the metadata file, b) update the metadata.

Parameters:
seemID the id of the metadata file describing the document
metadata an updated RDF-/Metadata file

Returns:
Returns true if the update was successful, else false.

Exceptions:
Exception Throws an exception in cases of an error.

String queryTest (String xml*QueryTest*) throws Exception
Performs a test query on one connected repository.
This is used to identify and measure any problems in the communication between the layers/nodes.

Parameters:
xmlQueryTest- xml document with query information

Returns:
Xml document with query information
Exceptions:
Exception Throws an exception in cases of an error.

String getAuditingInfoBetweenDates (String *info_level*, int *layer*, Date *from*, Date *to*, String *userPublicKey*) throws Exception
Method to retrieve the transaction information log generated in a specific layer between dates

Returns:
This method returns an xml nodelist of log entries as defined above.

Parameters:
from Date from to retrieve transaction information logs
to Date up to retrieve transaction information logs
info_level Info level of logs to retrieve: DEBUG, INFO, WARN or ERROR
layer Layer from retrieve information logs. Values:
0: Application Layer
1: Service
2: Distribution

3: Repository
userPublicKey Transaction id associated to request

Exceptions:
Exception Throws an exception in case of an error

String getRecursivelyAuditingInfoBetweenDates (String *info_level*, int *layer*, Date *from*, Date *to*, String *userPublicKey*) throws Exception

Method to retrieve the transaction information log generated in a specific layer between dates and its underlying layers

Parameters:
from Date from to retrieve transaction information logs
to Date up to retrieve transaction information logs
info_level Info level of logs to retrieve: DEBUG, INFO, WARN or ERROR
layer Layer from retrieve information logs. Values:
0: Application Layer
1: Application, Service
2: Application, Service, Distribution
3: Application, Service, Distribution, Repository
userPublicKey Transaction id associated to request

Returns:
This method returns a xml nodelist of log entries as defined above.

Exceptions:
Exception Throws an exception in case of an error

6.2.3 Registry-connector component

The list of methods is the same as in 6.2.6 since it implements all abstract Methods of the AbstractConnector-class used for querying data. Registries will normally be used by companies that manage the registry data with their existing software solutions. This component will therefore focus on accessing this data to make it available in the SRRN, while changing the data can be done with existing software solutions.

6.2.4 Distributor component

The list of methods is the same as in 6.2.6 since it implements all abstract Methods of the AbstractConnector-class.

6.2.5 Configuration component

The configuration of each component will be managed by this component. It therefore provides Methods to query all settings. A detailed overview about the methods can be found in the API documentation because of space-limitations. The following configuration is an example for data managed for a single Core Repository Layer instance accessed by the Repository Connector: For an explanation please read the comments within the text.

```
<repository>
	<name>SEAMLESS-Test Repository</name>
	<description>This is a test repository...</description>
	<availability>10</availability>
	<!-- 10=highly available server,
		1="testserver, don't store important data here" -->
	<permissions>
		<query>true</query> <!-- Can the server be queried? -->
```

Project co-funded by the European Commission within the
Sixth Framework Programme, Information Society Technologies (IST)

```
            <write>true</write> <!-- Can it be written? -->
            <allowMetadata>false</allowMetadata>
            <!—- Is it allowed to store metadata here? -->
            <allowBinary>true</allowBinary>
            <!—- Is it allowed to use this repository to store documents and other binary
            information? -->
      </permissions>
      <authentification>
            <!—- This is used to store security-related info -->
      </authentification>
      <connection>
            <serveraddress>rep.seamless.net:1234</serveraddress>
            <type>SeemSeed_webservices</type>
            <!-- We only have a webservice-Interface now but we
            better prepare for future technologies. -->
            <version>1.0</version>
      </connection>
</repository>
```

6.2.6 Interface to Service Layer

This is the 'main' webservice and is the connector between the Distribution Layer and the Service Layer. Its purpose is to exchange RDF based metadata, SPARQL-queries and binary files

DataHandler getDocument (String *seemID*) throws Exception

This method can be used to retrieve a whole document or other information which is described by a metadata file. This feature should only be used for small files (<1MB), and for large files the inclusion of a (mandatory) URL in each metadata entry will allow direct access. This is because of performance and traffic-issues.

Parameters:

seemID The ID of a binary file

Returns:

The file (e.g.. a PDF-document).

Exceptions:

Exception Throws an exception in cases of an error.

DataHandler getDocumentByURL (String *url*) throws Exception

This method can be used to retrieve a document by URL.
This URL can be located outside the SRRN and can point to any file available online. The URL may point to a web server or to an entry in a repository which is accessible via URLs. URLs might be included in a metadata file of a document. Please make sure that this is only used for small files. Every metadata file will contain a URL for direct access. This should be used for large files and for performance and traffic-issues. Please note: This might throw an exception in case of an inaccessible link (e.g. a https-link with an expired certificate)

Parameters:

url A URL (normal HTTP). This might be part of a metadata description

Returns:

The file (e.g. a PDF-document).

Exceptions:

Exception Throws an exception in cases of an error.

String getMetadata (String *seemID*) throws Exception

This method can be used to retrieve a metadata file based on the seemID.

Parameters:

seemID The ID of a metadata/RDF file

Returns:

The file. If no file exists then null will be returned

Exceptions:

Exception Throws an exception in cases of an error

String ping () throws Exception

This sends the IP address of the requester.
This method can be used to check if this Webservice is available and working.

Returns:

the IP address of the remote host

Exceptions:

Exception Throws an exception in cases of an error.

boolean queryAsync (String *query*, *long timestamp*, String *queryID, String nodelist*) throws Exception

The search is performed in exactly the same way as in the querySync method but this time the results of the search will not be collected and forwarded as a list, but they will be sent back as soon as they arrive (asynchronously) For example: A query finds 10 documents in 3 blocks (5 from the first repository, 3 from the distributed network and 1 from a UDDI-registry) and since all results are located on different systems in the distributed network they will most likely all be received by the Core Registry Layer at different times. In this case it will call the queryResult-Method from the RR Service Level 3 times (once for each result-block as soon as it arrives at the Core registry layer)

Parameters:

query the query used to query metadata/RDF files. This query is done in SPARQL
queryID an ID of this search, which will be forwarded to the queryResult-Method
timestamp The timestamp after which the distribution Layer will not process results anymore from the connected Distribution/Repository Layers
queryID An ID used to identify the query. It can be a random string and will be used as a parameter in the queryResult
nodelist xml structure defining the nodes that are allowed and not allowed to use for the query.

Returns:

Returns true if the search was started successfully. The results will be delivered asynchronously to the queryResult-Method

See also:

getRDF
queryAsync

Exceptions:

Exception Throws an exception in cases of an error.

boolean queryAsyncReturnContent (String *query*, long timestamp, String *queryID*) throws Exception

The search is performed in exactly the same way as in the querySyncReturnContent method but this time the results of the search will not be collected and forwarded as a list, but they will be sent back as soon as they arrive. For example: If this query finds 10 documents in 3 blocks (5 from the first repository, 3 from the distributed network and 1 from a UDDI-registry). In this case it will call the ReturnContent-Method from the RR Service Level 3 times (once for each result-block as soon as it arrives at the Core registry layer)

Parameters:

query the query used to query metadata/RDF files. This query is done in SPARQL
timestamp The timestamp after which the distribution Layer will not process results anymore from the connected Distribution/Repository Layers
queryID An ID used to identify the query. It can be a random string and will be used as a parameter in the queryResult

Returns:

Returns true if the search was started successfully. The results will be delivered asynchrounously to the queryResultReturnContent-Method

See also:

getRDF
queryAsync

Exceptions:

Exception Throws an exception in cases of an error.

boolean removeEntry (String *seemID*) throws Exception

Removes an Entry from the registry/repository. This can be used to remove documents or metadata
ATTENTION: If you wish to remove a document AND the metadata describing the document, then you will have to call this method twice: Once to remove the metadata and once to remove the document, described by the metadata.

Parameters:

seemID the id of the Entry to be removed

Returns:

returns true if the entry was removed successfully, else false.

Exceptions:

Exception Throws an exception in cases of an error.

boolean removeEntry (String *seemID*) throws Exception

Removes an Entry from the registry/repository. This can be used to remove metadata and the documents described by the metadata.

Parameters:

seemID the id of the Entry to be removed

Returns:

returns true if the entry was removed successfully, else false.

Exceptions:
Exception Throws an exception in cases of an error.

String storeDocument (DataHandler *document*, String *metadata*) throws Exception
Stores a document together with a describing metadata file.

Parameters:
document the file that should be stored (e.g. an PDF file,...)
metadata the metadata describing the file in RDF

Returns:
Returns an ID to access the metadata. Furthermore, the Metadata will contain an ID of the document. ID is unique within SRRN

Exceptions:
Exception Throws an exception in cases of an error.

String storeMetadata (String *metadata*) throws Exception
Stores an RDF file (e.g. Metadata file).
This is useful to store RDF files / Metadata files with no documents attached. Those RDF-files will only need to have 2 or 3 defined SEEMseed properties and have no other limitations. They can be used to store e.g. classifications and they can be queried directly.

Parameters:
metadata an RDF-/Metadata file to store. This can have additional information and will be searchable

Returns:
Returns an ID to access the metadata.

Exceptions:
Exception Throws an exception in cases of an error.

boolean updateDoc (String *seemID*, DataHandler *document*, String *metadata*) throws Exception
Updates a document and its metadata This method will a) catch the metadata file, b) get the associated document, c) update the document, d) add the documents id to the metadata e) update the metadata.

Parameters:
seemID the id of the metadata file describing the document
document an updated document (e.g. an PDF file,...)
metadata an updated RDF-/Metadata file

Returns:
Returns true if the update was successful, else false.

Exceptions:
Exception Throws an exception in cases of an error.

boolean updateMetadata (String *seemID*, String *metadata*) throws Exception
Updates an RDF file/metadata This method will a) catch the metadata file, b) update the

metadata.

Parameters:
seemID the id of the metadata file describing the document
metadata an updated RDF-/Metadata file

Returns:
Returns true if the update was successful, else false.

Exceptions:
Exception Throws an exception in cases of an error.

String [] QueryAsyncAccessLog(string transactionId, string userPublickey, string masterseemId, string timestamp)

This method performs a query for access information for a document stored on the SRRN and returns an XML document with the access information. The query is distributed to all connected Distribution Layers, all connected ebXML/UDDI registry layers and all connected Repository Layers.

Parameters:
transactionId – The transaction id associated to the query.
userPublickey – The users digital key.
masterseemId – The id for the documents stored on the SRRN.
Timestamp – The timestamp after which the query should be ignored.

Boolean [] QuerySyncAccessLogReturnContent(string transactionId, string userPublickey, string masterseemId, string timestamp)

This method performs a query for access information for a document stored on the SRRN and returns an XML document with the access information.

Parameters:
transactionId – The transaction id associated to the query.
userPublickey – The users digital key.
masterseemId – The id for the documents stored on the SRRN.
Timestamp – The timestamp after which the query should be ignored.

6.2.7 Interface to other Distribution Layers

The ‘interface to other distribution layer’ component uses the methods as defined by the abstract-Connector-Class and are described in the paragraph 6.3.6 ‘Interface to the service layer’

6.3 Repository Layer

6.3.1 Controller component

The controller component uses the same interfaces as described in the paragraph 6.3.6

6.3.2 RDF composer / decomposer

override string CreateEntry (string srcContent)

This method is used to store (create) a new metadata entry.
The provided RDF string is converted to the suitable format so that it can be stored in the metadata storage. After that, the necessary calls are made in order to create a new entry with the provided content in the metadata storage. The storage will generate an

identifier for the new entry. This identifier will be returned to the caller.

Parameters:

srcContent - the RDF string that should be stored

Returns:

The *unique identifier* of the newly created entry

Exceptions:

SEEMRepositoryExcetion - this is thrown in case of an error; it specifies the reason of the error.

Implements:

StorageAdapterBase

override string GetEntryContent (string seemID)

This method is used to retrieve the content of a metadata entry.
The necessary calls are made in order to retrieve the content of the metadata entry having the specified identifier. The retrieved content if converted to RDF and it is returned to the caller.

Parameters:

seemID - the identifier of the metadata entry that has to be retrieved

Returns:

An *RDF string* representing the content of the desired metadata entry, or *null* if the no metadata entry with the specified identifier could be found.

Exceptions:

SEEMRepositoryExcetion - this is thrown in case of an error; it specifies the reason of the error.

Implements:

StorageAdapterBase

override bool UpdateEntry (string seemID, string srcContent)

This method is used to change the content of a metadata entry.
The provided RDF string is converted to the suitable format so that it can be stored in the metadata storage. After that, the necessary calls are made in order to replace the content of the entry having the specified identifier with the newly provided content.

Parameters:

seemID - the identifier of the metadata entry that has to be replaced
srcContent - the RDF string that should be stored

Returns:

True if the operation could be performed, or *false* if the no metadata entry with the specified identifier could be found.

Exceptions:

SEEMRepositoryExcetion - this is thrown in case of an error; it specifies the reason of the error.

Implements:

StorageAdapterBase

override bool RemoveEntry (string seemID) [virtual]

This method is used to remove a metadata entry.
The necessary calls are made in order to remove the metadata entry having the specified identifier.

Parameters:

seemID - the identifier of the entry that has to be removed

Returns:

True if the operation could be performed, or *false* if the no metadata entry with the specified identifier could be found.

Exceptions:

SEEMRepositoryExcetion - this is thrown in case of an error; it specifies the reason of the error.

Implements:

StorageAdapterBase

6.3.3 SPARQL to SQL converter

string [] ApplySPARQL(string query)

This is the only public method exposed by the objects of this class.

The SPARQLquery string received as parameter is converted to SQL so that can it be used to query the relational database where the metadata is stored. As a limitation, the method accepts only queries that would return entry identifiers. I.e. the queries should always start with "select id".

Parameters:

query - the SPARQLquery that has to be applied on the metadata storage

Returns:

A collection of strings representing the *list of metadata entry identifiers* that are compliant to the conditions specified in the query.

6.3.4 File storage adapter

override string CreateEntry (string srcContent)

This method is used to store (create) a new document entry.
The provided content string is interpreted as the location of the binary file that has to be stored. The file is copied from the source location to the location provided in the configuration. After that, the necessary calls are made in order to create a new document entry with a reference to the location of the stored file. The storage will generate an identifier for the new entry. This identifier will be returned to the caller.

Parameters:

srcContent - the location of the file that should be stored

Returns:

The *unique identifier* of the newly created document entry

Exceptions:

RDFCDExcetion - this is thrown in case of an error; it specifies the reason of the error.

Implements:
StorageAdapterBase

override string GetEntryContent (string seemID)

This method is used to retrieve the content of a document entry.
The necessary calls are made in order to retrieve the file location of the document entry having the specified identifier. The retrieved file location is returned to the caller.

Parameters:
seemID - the identifier of the document entry that has to be retrieved

Returns:
A *string* representing the location of the file with the content of the desired document entry, or *null* if the no document entry with the specified identifier could be found.

Exceptions:
RDFCDException - this is thrown in case of an error; it specifies the reason of the error.

Implements:
StorageAdapterBase

override bool UpdateEntry (string seemID, string srcContent)

This method is used to change the content of a document entry.
The provided content string is interpreted as the location of the binary file that has to be stored. If a document entry with the specified identifier can be found, the new content file is copied from the source location to the location provided in the configuration and the old content file is deleted. After that, the necessary calls are made in order to update the specified document entry with the reference to the location of the new content file.

Parameters:
seemID - the identifier of the metadata entry that has to be replaced
srcContent - the location of the file that should be stored

Returns:
True if the operation could pe performed, or *false* if the no document entry with the specified identifier could be found.

Exceptions:
RDFCDException - this is thrown in case of an error; it specifies the reason of the error.

Implements:
StorageAdapterBase

override bool RemoveEntry (string seemed)

This method is used to remove a document entry.
If a document entry with the specified identifier can be found, the file refered by this entry is deleted. After that, the necessary calls are made in order to remove the document entry having the specified identifier.

Parameters:
seemID - the identifier of the entry that has to be removed

Returns:

True if the operation could be performed, or *false* if no document entry with the specified identifier could be found.

Exceptions:
SEEMRepositoryExcetion - this is thrown in case of an error; it specifies the reason of the error.

Implements:
StorageAdapterBase

6.3.5 Configuration component

string GetConfigValue (string name)

This method allows to retrieve the value of a configuration property.

Parameters:
name - the name of the property

Returns:
The *value* of the specified parameter, or *null* if no property with the specified name can be found.

Exceptions:
SEEMRepositoryExcetion - this is thrown in case of an error; it specifies the reason of the error.

bool SetConfigValue (string name, string value)

This method allows to set the value of a configuration property.
If no property with the given name exists, it is created.

Parameters:
name - the name of the property
value - the value of the property

Returns:
True if the operation succeeded, otherwise *false*.

Exceptions:
SEEMRepositoryException - this is thrown in case of an error; it specifies the reason of the error.

6.3.6 Interface to Distribution Layer

This is the webService interface exposed to the Distribution Layer. Its purpose is to exchange RDF based metadata, SPARQL-queries and binary files.

string ping ()

This method can be used to check the status of the webservice.
The system retrieves the IP address of the server and sends it back to the caller.

Returns:
The *IP address* of the machine where the webservice is running

Exceptions:
Error - Throws an exception in case of an error

string megaping (string transactionID, string userPublicKey)

This method can be used to check the status of all registered webservices.

Parameters:

transactionID TransactionID of megaping query
userPublicKey

Returns:

xml document with mega ping query information

Exceptions:

Error - Throws an exception in case of an error

bool canStoreMetadata ()

This method can be used to check if the repository can be used to store metadata. The system checks in the configuration if it is meant to support the storage of metadata entries.

Returns:

True if the repository can store metadata entries

Exceptions:

Error - Throws an exception in case of an error

bool canStoreBinary ()

This method can be used to check if the repository can be used to store binary files. The system checks in the configuration if it is meant to support the storage of binary files.

Returns:

True if the repository supports binary files storing

Exceptions:

Error - Throws an exception in case of an error

bool canStoreBinaryByName (string documentName)

This method can be used to check if the repository can be used to store a binary file with the specified name.
The system checks in the configuration if it is meant to support the storage of binary files. If this is supported, the system will check also if there are limitations on the file types that can be stored, based on the extension of the file name.

Parameters:

documentName - the name of the file that should be stored

Returns:

True if the repository could store a binary file with the specified name

Exceptions:

Error - Throws an exception in case of an error

string storeMetadata (string metadata)

This method is used to store metadata entries in the repository.

The format of the metadata is RDF. When creating the new entry, the repository generates a unique identifier for it. This identifier is returned to the caller. The stored metadata can be queried by calling the querySync method via SPARQLqueries. A metadata entry can be retrieved by calling the getMetadata method.

Parameters:
metadata - a string containing the metadata that should be stored in RDF format

Returns:
The *unique identifier* of the stored metadata entry

Exceptions:
Error - Throws an exception in case of an error
NotSupported - The storage of the metadata is not supported

string storeDocument (string documentName)

This method is used to store documents (*attached* binary files) in the repository. The content of the document is provided as an *attachment* to the message. When creating the new entry, the repository generates a unique identifier for it. This identifier is returned to the caller. Only one attachment per message is supported. If no attachment is found, this is considered an error and an exception is thrown. No queries are supported on the stored documents. The content of the document can be retrieved by calling the getDocument method.

Parameters:
documentName - the name of the file where the document content should be stored

Returns:
The *unique identifier* of the stored document

Exceptions:
Error - Throws an exception in case of an error
NotSupported - The storage of the documents (binary files) is not supported

string [] querySync (string query, long timestamp)

Performs a query on the metadata stored in the repository.
The queries are expressed in SPARQLformat.

Parameters:
query - the query used to query metadata. This query is done in SPARQL
timestamp - the timestamp after which no results should be returned.

Returns:
An *array of IDs of metadata entries* that match the query. These IDs can then be used to access the corresponding metadata using the getMetadata-method

Exceptions:
Error - Throws an exception in case of an error

string getMetadata (string seemID)

Retrieve a metadata entry based on its ID.
The metadata with the specified ID is retrieved from the storage and returned as an RDF string to the caller.

Parameters:

seemID - the identifier of the desired metadata entry

Returns:

An *RDF string* containing the retrieved metadata entry, or an empty string if no metadata entry with the specified identifier could be found in the storage.

Exceptions:

Error - Throws an exception in case of an error

bool getDocument (string seemID)

Retrieve a document (binary file) based on its ID.
The document content is *attached* to the response message.
The document (binary file) with the specified ID is retrieved from the storage and returned to the caller as an *attachment*.

Parameters:

seemID - the identifier of the desired document

Returns:

True if a document with the specified identifier could be found in the storage. *The content of the file is attached to the response message*. If the returned value is false, the response message will have no attachment.

Exceptions:

Error - Throws an exception in case of an error

bool updateMetadata (string seemID, string metadata)

Updates the metadata entry specified by the provided identifier.
The content of a metadata entry that is already stored in the repository can be updated by providing the identifier of the entry and the new content as parameters.

Parameters:

seemID - the identifier of the desired metadata entry
metadata - a string containing the new metadata that should be stored, in RDF format

Returns:

True if the update was successful; the only case when the update could fail without throwing an exception is when the specified entry is not found

Exceptions:

Error - Throws an exception in case of an error

bool updateDocument (string seemID)

Updates the document specified by the provided identifier (the document content is *attached* to the message).
The content of a document that is already stored in the repository can be updated by providing the identifier of the document as parameter, and the new content *as attachment*. If no attachment is found, this is considered an error and an exception is thrown.

Parameters:

seemID - the identifier of the desired document

Returns:

True if the update was successful; the only case when the update could fail without throwing an exception is when the specified document is not found

Exceptions:

Error - Throws an exception in case of an error

bool removeEntry (string seemID)

Removes the entry (document or metadata) specified by the provided identifier.
This is the only way an entry (metadata or binary file) can be removed from the repository. The system makes no check on the consistency of metadata referring to the deleted entry. This falls in the responsibility of the caller.

Parameters:

seemID - the identifier of the desired entry

Returns:

True if the removal was successful; the only case when the removal could fail without throwing an exception is when the specified entry is not found

Exceptions:

Error - Throws an exception in case of an error

string [] queryTest (string xmlQueryTest)

Performs a test query on the repository.
This function is used to perform a test query on a single repository to identify any problem in the communication between all layers/nodes. The Repository Layer doesn't perform any query to the database but update the xmlQueryTest structure and return to the Distribution Layer.

Parameters:

xmlQueryTest – xml document with data on all passed nodes

Returns:

XML document with data on all passed nodes

Exceptions:

Error - Throws an exception in case of an error

String [] QuerySyncAccessLog(string transactionId, string userPublickey, string masterseemId, string timestamp)

This method performs a query for access information for a document stored on the SRRN and returns an XML document with the access information.

Parameters:

transactionId – The transaction id associated to the query.
userPublickey – The users digital key.
masterseemId – The id for the documents stored on the SRRN.
Timestamp – The timestamp after which the query should be ignored.

String getAuditingInfoBetweenDates (String *info_level*, int *layer*, Date *from*, Date *to*, String *userPublicKey*) throws Exception

Method to retrieve the transaction information log generated in a specific layer between dates

Returns:
This method returns an xml nodelist of log entries as defined above.

Parameters:
from Date from to retrieve transaction information logs
to Date up to retrieve transaction information logs
info_level Info level of logs to retrieve: DEBUG, INFO, WARN or ERROR
layer Layer from retrieve information logs. Values:
0: Application Layer
1: Service
2: Distribution
3: Repository
userPublicKey Transaction id associated to request

Exceptions:
Exception Throws an exception in case of an error

String getRecursivelyAuditingInfoBetweenDates (String *info_level*, int *layer*, Date *from*, Date *to*, String *userPublicKey*) throws Exception

Method to retrieve the transaction information log generated in a specific layer between dates and its underlying layers

Parameters:
from Date from to retrieve transaction information logs
to Date up to retrieve transaction information logs
info_level Info level of logs to retrieve: DEBUG, INFO, WARN or ERROR
layer Layer from retrieve information logs. Values:
0: Application Layer
1: Application, Service
2: Application, Service, Distribution
3: Application, Service, Distribution, Repository
userPublicKey Transaction id associated to request

Returns:
This method returns an xml nodelist of log entries as defined above.

Exceptions:
Exception Throws an exception in case of an error

6.3.7 WSDL

Will be updated once available.

7 Conclusions

By reusing the SRRN of the SEEMseed project a solid base has been used and the improvements in this distributed federated P2P network will make it performing and suitable for the applications and the ontology services of the SEAMLESS project.

All implementation partners of WP3.3 participated in this document or reviewed it and have found this document to be a solid base to perform the implementation of the SEAMLESS distributed storage system.