Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Förderagentur für Innovation KTI

SciMantic

Deliverable D3.1

Implementation and Evaluation of the SciMantic's Semantic Content Infrastructure

The SciMantic Consortium

University of Zurich
Trialox AG

For more information on this document or the SciMantic project, please contact:

Prof. Dr. Burkhard Stiller
University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14
CH-8050 Zürich
Switzerland

Phone: +41 44 635 67 10
Fax: +41 44 635 68 09
E-mail: stiller@ifi.uzh.ch

Document Control

Title:	Implementation and Evaluation of the SciMantic's Semantic Content Infrastructure
Type:	R&D
Editor:	Hasan
E-mail:	hasan@ifi.uzh.ch
Author:	Hasan
Contributors:	Daniel Spicar
Delivery Date:	18.07.2011

Executive Summary

The technical results of SciMantic have been presented in D2.1 (Functional Requirements and Analysis of Mechanisms for a Semantic Content Infrastructure) and D2.2 (Architecture of a Secure and Scalable Semantic Content Infrastructure). This deliverable, D3.1 (Implementation and Evaluation of the SciMantic's Semantic Content Infrastructure), is the last technical deliverable in this series which completes the design work in the previous deliverable with implementation and evaluation of the SciMantic's Semantic Content Infrastructure (SCI).

Although some refinements in the implementation to achieve modularity and better performance have been made, the implementation has principally been straightforward according to the design. This deliverable does not only focus on the API descriptions of services provided by the SCI, but also on the service implementation, where necessary, to enable developers to understand mechanisms used and potentially extend or improve the SCI implementation in future. The SCI implementation consists basically of OSGi bundles which can be divided into the following categories (groups): *third party bundles*, *Apache Clerezza bundles*, and *SciMantic specific bundles*. Developers implementing application logic writes *application bundles* which use services and functionality offered by the other bundles. The major part of implementation description in this deliverable is dedicated to Apache Clerezza bundles and SciMantic specific bundles. Apache Clerezza bundles are the result of development collaboration between SciMantic partners which see a potential in pushing further development of the resulting software by an open source community under Apache Software Foundation. SciMantic specific bundles are the results of an effort to apply Peer-to-Peer (P2P) technology in a content sharing environment which bases on Semantic Web for modeling linked distributed data. The Knowledge Sharing System (KSS) scenario defined in D2.1 has been prototypically implemented to show the benefit of combining P2P and Semantic Web technology.

The results of the performance and scalability evaluation show that the KSS, implemented on top of the Apache Clerezza, is able to deliver a good response time in the range of a few hundreds of milliseconds to one second for several hundreds of semi concurrent accesses to its Web services. For this evaluation, tests were set up with negligible network delay by directly connecting the client with the server under test. Furthermore, the use of Distributed Hash Table (DHT) for storing keyword-based index of knowledge units allows for an efficient keyword-based search. The time needed to insert or retrieve a knowledge unit does not depend on the number of knowledge units already present in the DHT. However, it does grow with the growing number of nodes participating in the knowledge sharing network. Within the evaluated range of the number of participating nodes, the resulting insertion or retrieval time seems to increase linearly, if not logarithmically. Therefore, it can be concluded that the prototypical implementation of the KSS is feasible to be applied to content sharing scenarios where the Web services are not computationally intensive, and the sharing network may comprise up to several thousands of participating nodes.

Table of Contents

1 Introduction 1
2 Third Party Bundles 3
3 Apache Clerezza Bundles 4
3.1 Smart Content Binding 5
3.1.1 SCB Core 5
3.1.2 Ontologies 16
3.1.3 SCB Utilities 17
3.1.4 Jena Façade 20
3.1.5 Implemented WeightedTcProviders 20
3.1.6 Implemented Parsing and Serializing Providers 21
3.1.7 Jena-based SPARQL Engine 22
3.1.8 Composite Resource Indexing Service (CRIS) 22
3.1.9 Web Interface 25
3.2 Triaxrs 26
3.2.1 Implemented Entity Providers 26
3.2.2 Supported Context Types 28
3.2.3 Bundle Path Prefix 28
3.3 Static Web 28
3.4 Platform 29
3.4.1 Information Resource Modeling 29
3.4.2 Platform TripleCollections 31
3.4.3 Extended Content Graph and ContentGraphProvider Service 31
3.4.4 GraphNodeProvider Service 31
3.4.5 PageNotFoundService 32
3.4.6 Type Handling 32
3.4.7 Type Rendering 34
3.4.8 Authentication 38
3.4.9 Authorization 40
4 SciMantic Specific Bundles 42
4.1 Knowledge Unit Identifier 42
4.2 Ontology 43
4.3 Knowledge Unit Management Service 43
4.4 Knowledge Unit Tagging Service 44
4.5 Knowledge Unit Sharing Service 44
4.6 Knowledge Unit Searching Service 45
4.7 Knowledge Unit Update Event Service 45
4.8 Knowledge Unit Publisher Discovery Service 47
4.9 DHT Service 47
5 Evaluation 49
5.1 Performance Analysis of Web Interfaces 49
5.2 Performance Analysis of the KSS DHT Service 49
5.3 Scalability Evaluation 50
6 Summary 53

1 Introduction

Based on the architecture design described in D2.2 [6], a Semantic Content Infrastructure (SCI) has been prototypically implemented and evaluated which is documented in this deliverable D3.1. The SCI comprises a network of interacting nodes which may join and leave the network at any time. Each SCI node implements an instance of a Semantic Content Sharing System (SCSS) and provides Web services which are accessible in a *secured* way: users are *authenticated* and their actions must pass an *authorization* process. Figure 1 depicts a high level overview of the implementation architecture of the SCSS. The SCSS is a Java program which integrates an OSGi Framework and a set of OSGi bundles to provide the required functionality. These OSGi bundles are activated within the execution environment provided by the OSGi Framework and they can be divided into 4 groups:

1. Third Party Bundles: OSGi bundles implemented by a third party to provide certain functionality. Examples include Triple Store implementation, Web Server, and Distributed Hash Table.
2. Apache Clerezza Bundles: OSGi bundles implemented by Apache Clerezza community. These bundles constitute a software platform to build RESTful Web Services for implementing business logic and workflows, managing and manipulating data stored as RDF graphs. These bundles can use functionality provided by Third Party Bundles.
3. SciMantic Specific Bundles: OSGi bundles implemented within the SciMantic project to add functionality to Apache Clerezza, concerning the sharing of distributed content. These bundles use functionality provided by Third Party and Apache Clerezza Bundles.
4. Application Bundles: OSGi bundles implementing the business logic which is application specific. These bundles can use functionality provided by all other bundles.

Services provided by Apache Clerezza and SciMantic Specific Bundles are Web Services and thus accessible through HTTP. Furthermore, HTTP is also used for communication among SciMantic Specific Bundles of different SCI Nodes. With respect to Application Bundles or Third Party

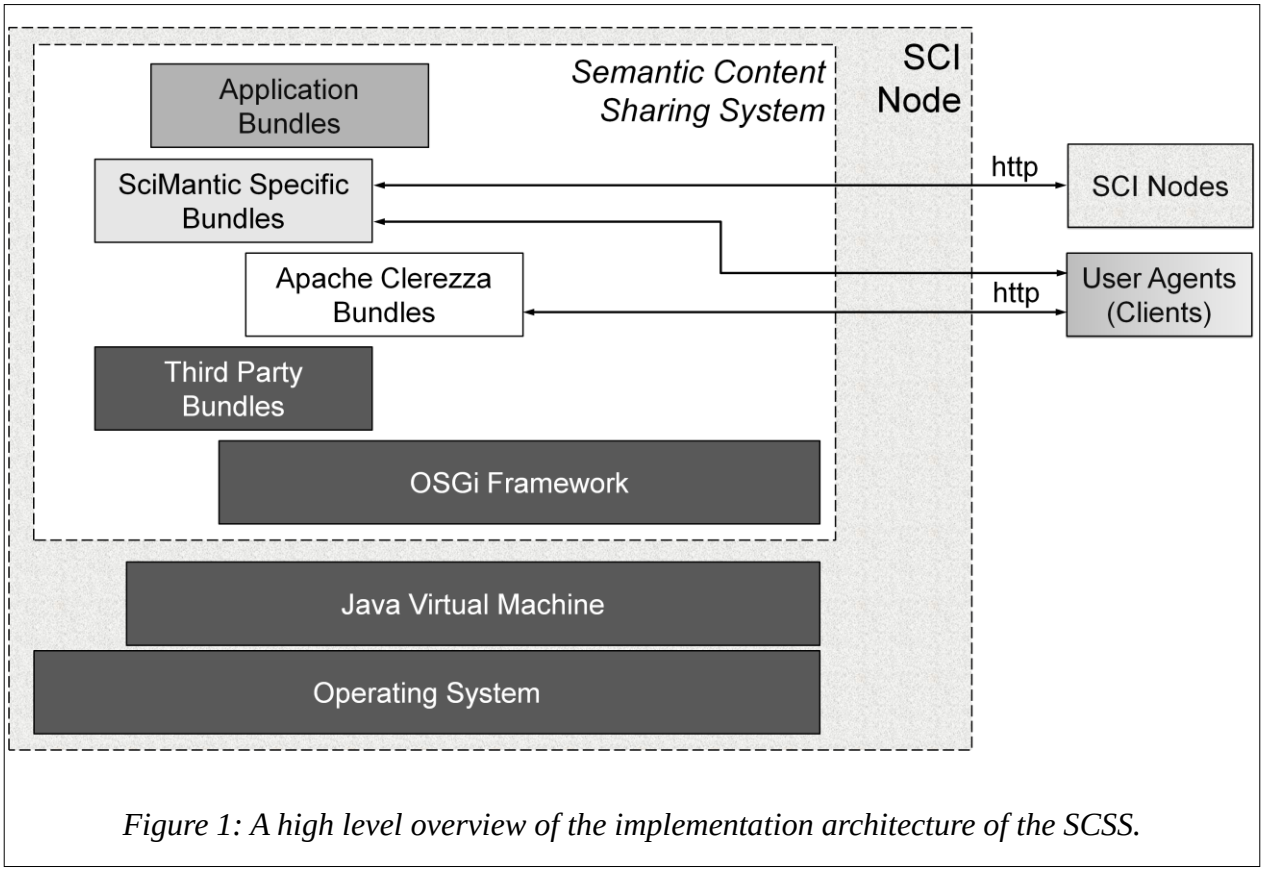

Figure 1: A high level overview of the implementation architecture of the SCSS.

Bundles, they may communicate with each other within or across SCI Node. However, this communication is transparent to Apache Clerezza Bundles and SciMantic Specific Bundles.

Currently, Apache Felix [17] is the OSGi Framework integrated into the SCSS. However, with minor modifications, it should also be possible to use other OSGi Frameworks as well, *e.g.*, Eclipse Equinox.

A Knowledge Sharing System (KSS) has been developed and is being improved and extended to serve as a demonstrator. Experiments are carried out on the demonstrator to evaluate the performance and scalability of the system.

The remaining sections in this deliverable are organized as follows: in Section 2, Third Party Bundles are described briefly, in particular regarding their functionality and interfaces. Section 3 and Section 4 describes the implementation of Apache Clerezza and SciMantic Specific Bundles respectively. In Section 5, the implementation of the KSS is presented, whereas Section 6 shows the evaluation results. Finally, a summary of the implementation and evaluation of the SCSS is given in Section 7.

2 Third Party Bundles

Third party bundles used within SciMantic shall have the following characteristics:

- Stable: widely used and well tested by the respective community
- Easily replaceable: various implementations of the same functionality exist with a well defined API

Third party bundles might not be available as OSGi bundles; in this case, an OSGi version will be created. If a third party bundle is supposed to be exchangeable and a standard API does not exist, a generic API will be defined, so that adaptors can be written which provide a uniform interface across different technologies. Table 1 lists main third party bundles which are, or in case of alternative solutions, can be used within SciMantic.

Table 1: Third Party Bundles.

Third Party Bundles	Main Function	Remarks
Jena	Triple Store Parser/Serializer	- Single machine - Jena API - Supports transactions - SPARQL support - Parser and serializer for various RDF formats
Jena TDB	Triple Store	- Single machine - Jena API - High performance - Non-transactional - SPARQL support
Sesame	Triple Store Parser/Serializer	- Single machine - Sesame API - Supports transactions - SPARQL support - Parser and serializer for various RDF formats
JSON.simple	Parser/Serializer	- Parser and serializer for JSON
Jetty	Web Server	
WRHAPI	Web Request Handler	- Modeling HTTP-Requests and Responses
Apache Lucene	Indexing	
openchord	Distributed Hash Table	

3 Apache Clerezza Bundles

Bundles implemented within Apache Clerezza are grouped as shown in Figure 2. These groups have the following functionality:

- **Smart Content Binding** bundles provide functionality for accessing and manipulating RDF graphs. These bundles implement Smart Content Binding described in Deliverable D2.2 [6].
- **Triaxrs** bundles implement JSR-311 (JAX-RS) specification, a Java API for RESTful Web services.
- **Utilities** bundles implement special functionality which is commonly needed by other bundles, especially by platform bundles.
- **Static Web** bundles provide access to a collection of static Web resources, in particular commonly needed JavaScripts or JavaScript Libraries, as well as some pre-defined styles and templates for rendering a Web page. Those JavaScripts or JavaScript Libraries may be developed by third parties.
- **Platform** bundles provide add-on functionality for (Web) applications to enforce user authentication and access control, process Web requests and render Web responses. They also comprise bundles that enable applications amongst others to configure the platform, store and edit application data in a special graph called Content Graph, and work with SKOS concepts.

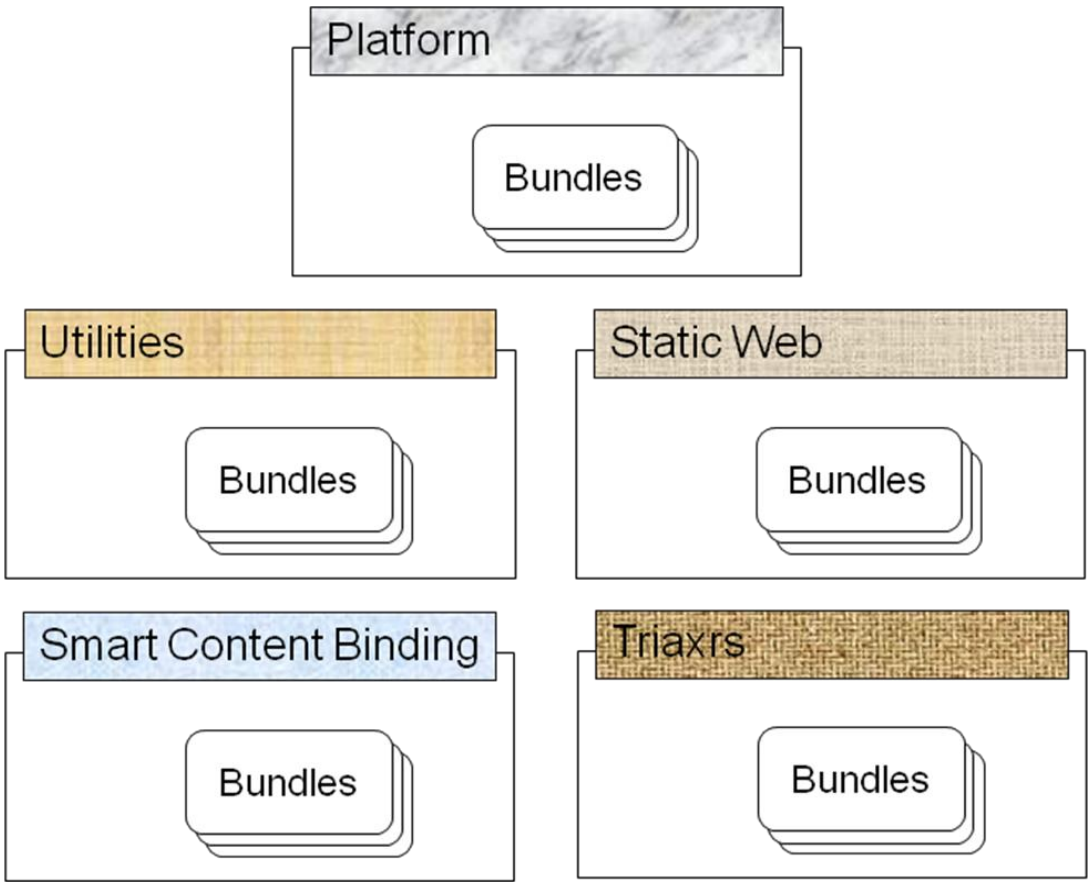

Figure 2: Apache Clerezza bundles.

3.1 Smart Content Binding

The Smart Content Binding bundles are depicted in Figure 3. The heart of this group of bundles is the SCB Core bundle which provides important interfaces for other bundles to access RDF graphs, manipulating their triples, serializing them to a specific format, and parsing serialized graphs of specific formats into an internal graph model which can then be persistently stored. The SCB Core

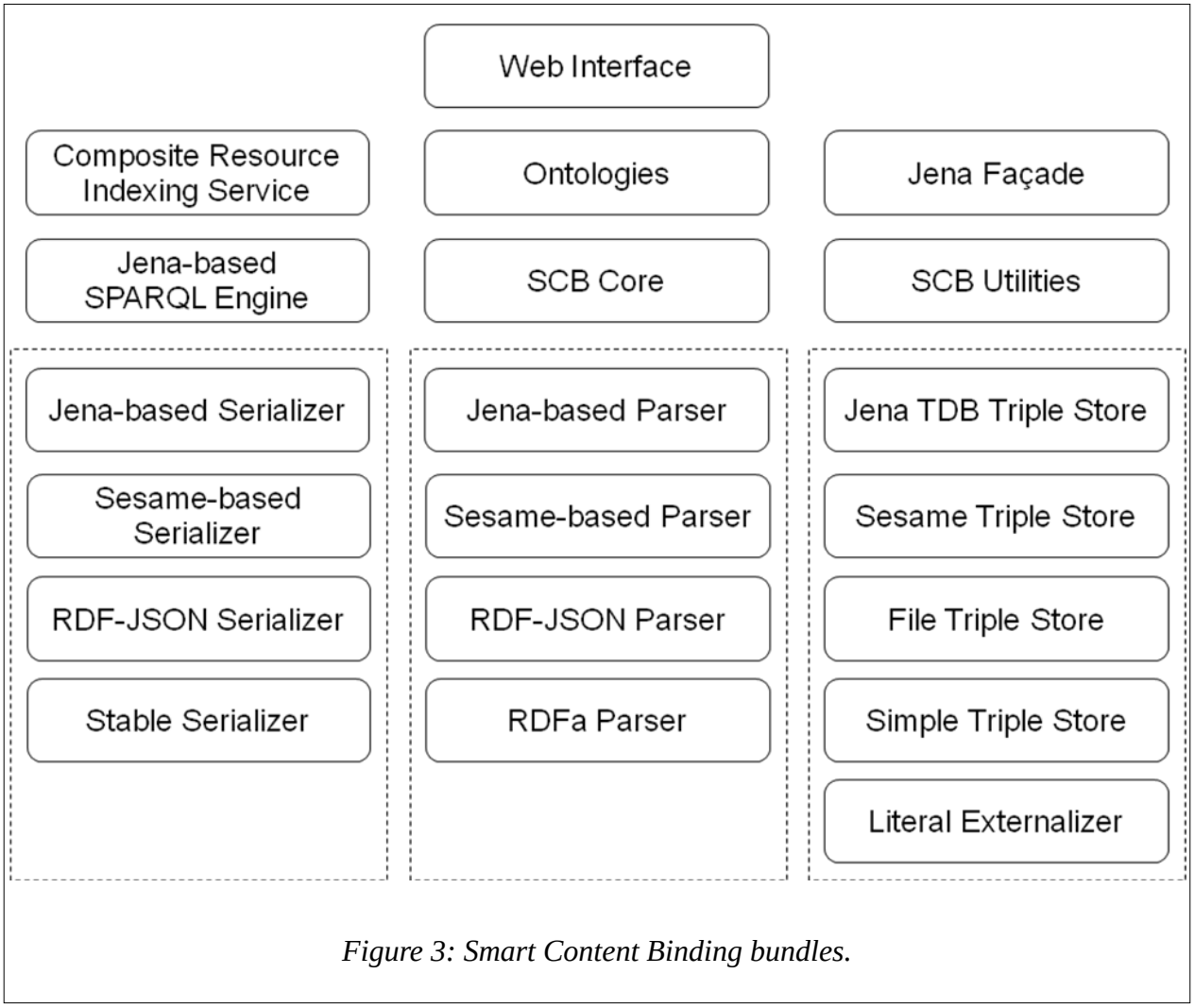

Figure 3: Smart Content Binding bundles.

bundle relies on services provided by other bundles within this group for performing technology specific tasks – amongst others – persistent storage, graphs parsing, and serialization. Furthermore, this group also includes a bundle which provides Java objects for representing classes and properties of defined by popular ontologies like OWL, RDF, RDFS, SKOS, DC, FOAF, etc. Finally, to allow Web access to certain Core services, the Web Interface bundle is developed. The following subsections describe each of the bundles in this group.

3.1.1 SCB Core

One of the key functions of the SCB Core is the modeling of the RDF graph. This SCB model aims at providing other bundles with a uniform access to technology specific RDF graphs. A graph consists of a set of nodes (vertices) and a set of edges (arcs) which connect pairs of nodes. **Error! Reference source not found.** depicts the hierarchy of node types in SCB. On top of the hierarchy is the Resource which basically can represent anything. It is either a Literal or a NonLiteral. A UriRef is a NonLiteral that represents an RDF URI Reference, which is a Unicode string as defined in [8]. Note that an RDF URI Reference is not the same as URI defined by RFC 3986. A BNode (blank

node) is also a NonLiteral which is just a unique node without an intrinsic name. Literals can be divided into PlainLiterals and TypedLiterals. A PlainLiteral represents a string which may have a language tag, whereas a TypedLiteral represents a value which has a data type.

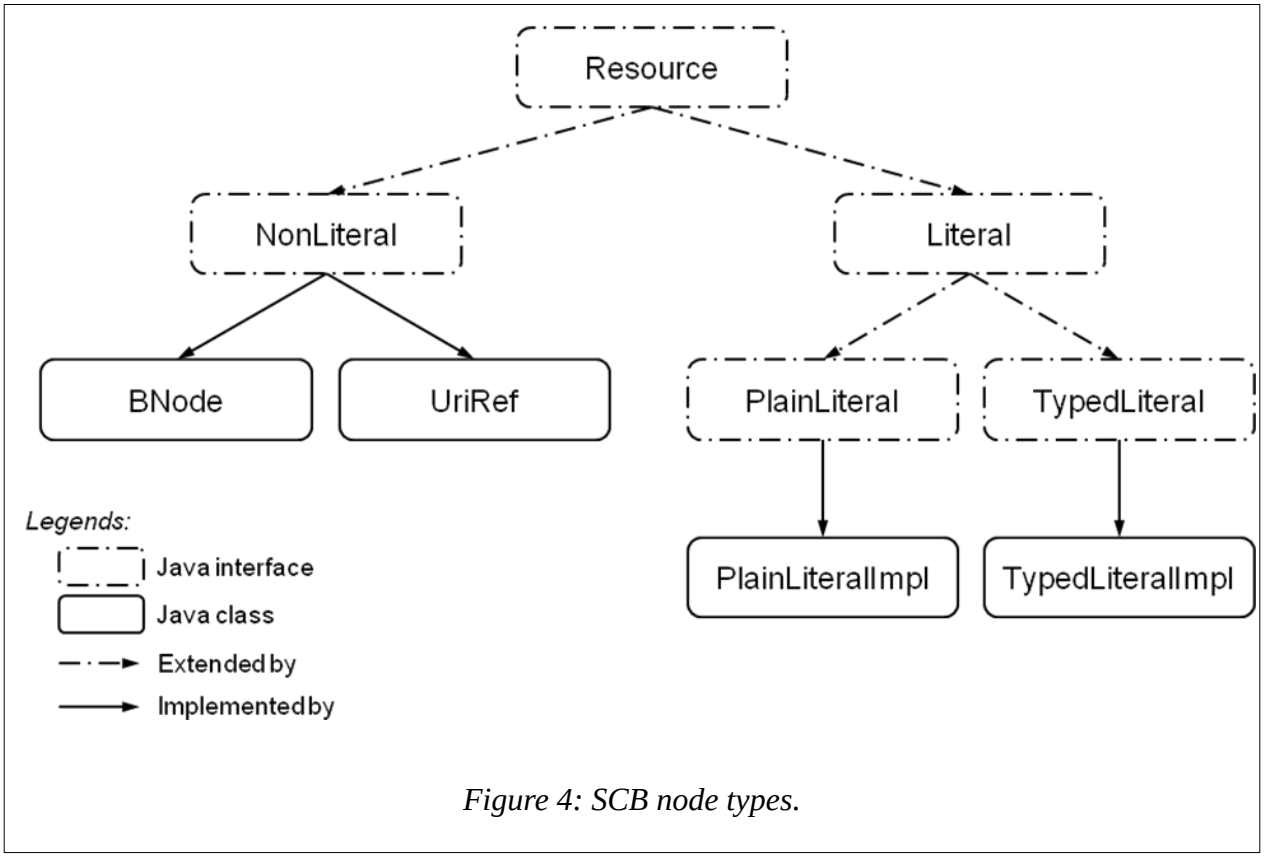

Figure 4: SCB node types.

The SCB data model of RDF graphs is shown in Figure 5. The SCB's TripleCollection is the highest abstraction level of a graph. It is a Java interface which extends java.util.Collection<Triple>. A Triple is a data structure containing a subject, a predicate, and an object (also known as a statement). The subject of a Triple must be a NonLiteral, the predicate must be a UriRef, and the object can be of any Resource type. Table 2 lists some main public methods which must be implemented by an implementation of a TripleCollection.

Table 2: Main public methods of TripleCollection.

Return Value	Method Signature and Description
`Iterator<Triple>`	`filter(NonLiteral subject, UriRef predicate, Resource object)` This method filters triples for a given pattern and returns an iterator for all Triples matching that specified pattern. Specifying null as parameter means that any value matches that parameter. filter(null, null, null) returns the same as iterator().
`void`	`addGraphListener(GraphListener listener, FilterTriple filter, long delay)` This method can be used to register a GraphListener, so that it gets notified if there are changes to the TripleCollection. However, the listener will only be notified if the Triple that was part of the modifiaction matches the specified FilterTriple. The notification will be delivered to the listener after the specified delay time (in milli-seconds) has passed. If more matching events occur during the delay period, then they are passed all together at the end of the delay period. If delay is 0, notification will happen immediately. If the listener unregisters or the platform is stopped within the delay period, then the already occurred events may not be delivered.

	All implementations must support this method. Implementations for immutable TripleCollection will typically provide an empty method, they shall not throw an exception. Implementations may also choose not to, or only partially propagate their changes to the listener. They should describe the behavior in the documentation of the class. Implementations should keep weak references to the listeners, so that the listener can be garbage collected if it is no longer referenced by another object.
`void`	`removeGraphListener(GraphListener listener)` Use this method to remove the specified GraphListener from the list of registered listeners. This listener will no longer be notified if the TripleCollection is modified.

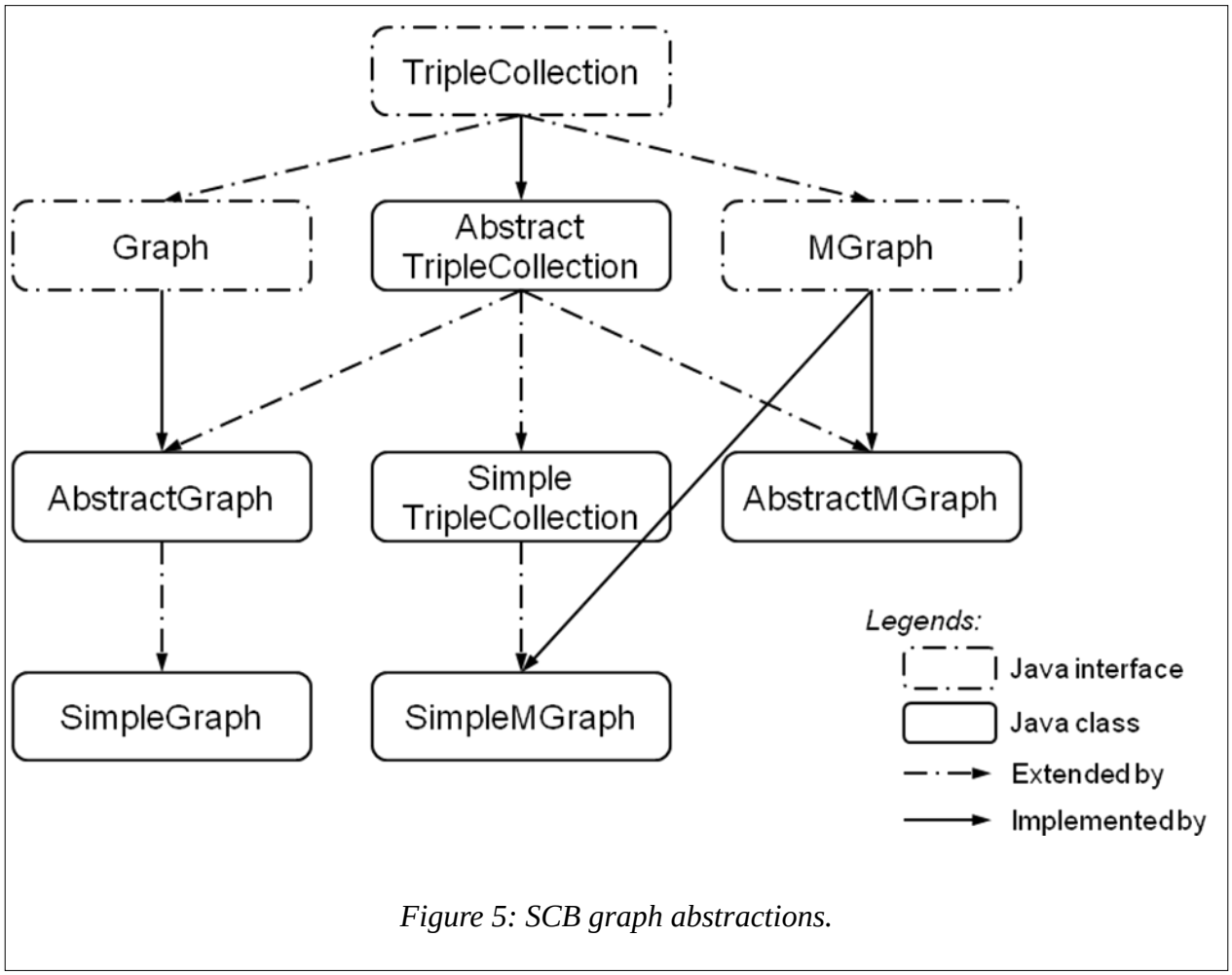

Figure 5: SCB graph abstractions.

Triples of a TripleCollection can be retrieved using the filter() method by specifying a pattern consisting of a subject, a predicate, and an object pattern. The pattern is either a specific value or null which matches to any value. Changes to a TripleCollection (Triples are added or removed) can be notified to a registered GraphListener. GraphListeners are registered through the method addGraphListener() and deregistered through the method removeGraphListener(). A class that implements the interface GraphListener must implement the method:

```
void graphChanged(List<GraphEvent> events)
```

The method graphChanged is invoked when the corresponding TripleCollection was modified, to which the GraphListener was registered. A list of GraphEvents is passed as argument. Each GraphEvent contains the following information: the Triple that was part of the modification, the type of modification (addition or removal), and the TripleCollection that was modified.

In registering a GraphListener, a FilterTriple is passed to check whether a Triple is of interest to the

listener. The constructor of the class FilterTriple receives a pattern for a subject, a predicate, and an object which can be a specific value or null. The match() method of FilterTriple will then be used to examine whether a Triple affected in the modification of the TripleCollection does match.

Table 3 describes subinterfaces of TripleCollection and their implementations.

Table 3: Subinterfaces of TripleCollection and their implementations.

TripleCollection	Description
AbstractTripleCollection	This abstract class implements the TripleCollection interface. It extends java.util.AbstractCollection<Triple> and overrides its methods iterator() and contains(). Since notification of TripleCollection modification is supported through the method filter(), add(), and remove(), those methods should not be overridden to keep this support. Instead, these three methods are available to be overridden if necessary: performFilter(), performAdd, and performRemove().
Graph	A Graph is modeled as a set of Triples. This interface does not extend java.util.Set because of the different identity constraints, *i.e.* two Graphs may be equal (isomorphic) even if their set of Triples are not. Implementations must be immutable and throw respective exceptions if add/remove-methods are called.
MGraph	An MGraph is mutable. Two MGraphs are equal if changes to any of the two MGraphs are reflected immediately in the other. It provides a getGraph() method to obtain a snapshot set of Triples within this MGraph.
AbstractGraph	AbstractGraph extends AbstractTripleCollection and is an abstract implementation of Graph. It implements the method hashCode() and equals() for checking graph isomorphism.
AbstractMGraph	AbstractMGraph extends AbstractTripleCollection and is an abstract implementation of MGraph. It implements the method getGraph().
SimpleTripleCollection	SimpleTripleCollection extends AbstractTripleCollection. This class is not public, thus applications should use SimpleGraph or SimpleMGraph. It has various constructors to initially fill the instance variable of type Set<Triple> with the set of Triples obtained through the constructor's argument. It is able to detect concurrent modification and throw the respective ConcurrentModificationException.
SimpleGraph	SimpleGraph extends AbstractGraph. Normally Triples are copied if a Graph is instantiated. This class provides a constructor which allows one to specify if the TripleCollection passed to the constructor might change in the future. If not, the collection isn't copied, but only its reference will be stored, thus making things more efficient.
SimpleMGraph	SimpleMGraph extends SimpleTripleCollection and implements MGraph. It overrides getGraph() to return a SimpleGraph.

Support of Concurrent Access Control

To avoid concurrent modification to an MGraph by two or more threads, a thread should acquire an access lock before it accesses the MGraph. Figure 6 depicts the interface LockableMGraph which extends MGraph and the class LockableMGraphWrapper which implements this interface. An implementation of a LockableMGraph must provide the method getLock() which returns a java.util.concurrent.locks.ReadWriteLock. This lock allows for creation of read and write-locks that span individual method calls. A thread having a read-lock prevents other threads from writing to, but not from reading the MGraph, whereas a thread having a write-lock prevents other threads from reading and writing to the MGraph. The LockableMGraphWrapper wraps an MGraph as a

LockableMGraph. The implementation of the locking support in LockableMGraphWrapper is based on java.util.concurrent.locks.ReentrantReadWriteLock. The wrapper *automatically* acquires the respective lock before a method of the wrapped MGraph is invoked. The Iterator<Triple> returns by some methods, *e.g.*, the filter() method, is a special Iterator called LockingIterator which takes care of locking the MGraph, when a thread is iterating through the Triples or removing current Triple.

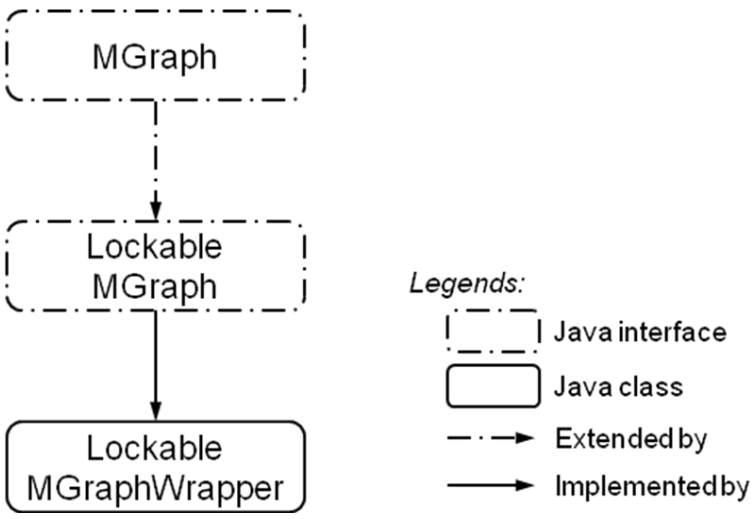

Figure 6: Lockable MGraph.

The LockableMGraphWrapper implements the method getLock() which should not be needed in most cases, because this class performs an auto lock. Therefore, a more suitable name for this class should be AutoLockMGraphWrapper. A LockableMGraphWrapper would just have to provide the lock and leave it to the thread when to invoke the lock. As implemented now, adding 100 Triples to an MGraph using LockableMGraphWrapper will lock and unlock the MGraph 100 times. The same remarks hold true for the LockingIterator which is more suitable be named AutoLockIterator. Iterating 100 Triples through the LockingIterator will lock and unlock the corresponding MGraph 100 times.

Support of Secure Accesses

In many cases it is necessary to control access to a TripleCollection. This means that a thread will be allowed access to a TripleCollection if the user that the thread represents owns the required permissions. Thus, to support access control to a TripleCollection, the following is needed:

- A name to identify a TripleCollection
- A Java class extending java.security.Permission to define permissions for accessing a named TripleCollection

In Apache Clerezza, the data type of the name used to identify a TripleCollection is a UriRef and the subclass of the java.security.Permission is called TcPermission. The constructor of the TcPermission requires a name pattern string and an actions string. A pattern matches the name of a TripleCollection if the pattern is equal to this name (its Unicode string to be precise), or if the pattern ends with "/*" and the name of the TripleCollection starts with the substring preceding the '*' in the pattern. The actions string is a comma separated list of the strings “read” and “readwrite”. The canonical form is just “read” or “readwrite”, because “readwrite” implies “read”.

SCB Core defines a Java class called TcAccessController which provides the following two methods to check whether the user executing the thread has the required permissions to access a specific TripleCollection:


```
public void checkReadPermission(UriRef tripleCollectionUri);
public void checkReadWritePermission(UriRef tripleCollectionUri);
```

Figure 7 depicts two types of TripleCollection wrappers which perform access control prior to each method invocation which accesses the wrapped TripleCollection. For example, before a Triple can be added to the wrapped TripleCollection, the readwrite permission is checked. The SecuredTripleCollection wraps a TripleCollection, whereas the SecuredMGraph wraps a LockableMGraph. The SecuredMGraph also implements the LockableMGraph interface, but delegates the getLock() method invocation to the wrapped LockableMGraph. In order to enable access control in SecuredTripleCollection and SecuredMGraph, two objects are needed: the name of the wrapped TripleCollection (respectively LockableMGraph) and an instance of TcAccessController. These two objects are passed to the wrappers through their constructor:

```
public SecuredTripleCollection(TripleCollection wrapped, UriRef name,
        TcAccessController tcAccessController);
public SecuredMGraph(LockableMGraph wrapped, UriRef name,
        TcAccessController tcAccessController);
```

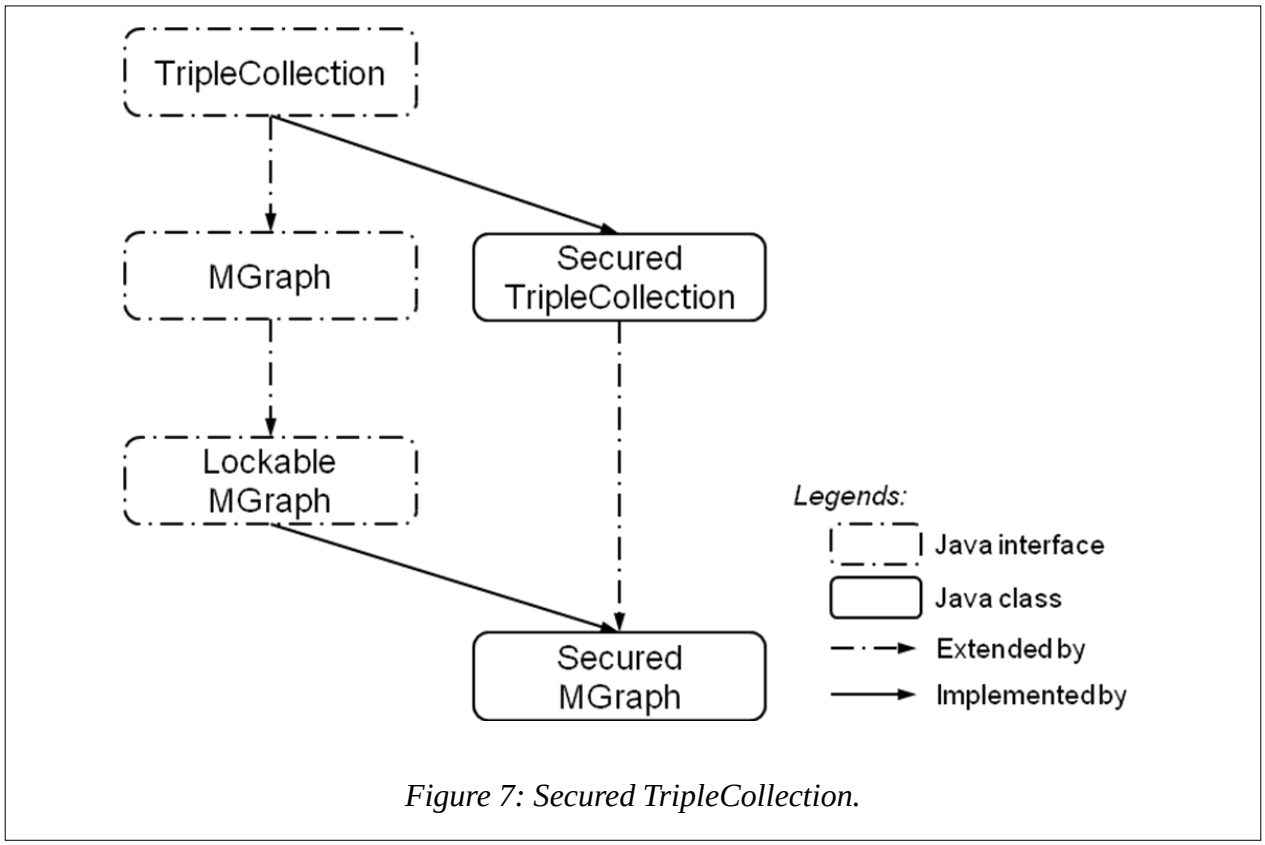

Figure 7: Secured TripleCollection.

The Iterator<Triple> returned by the filter() method of the secured TripleCollection also performs access control when used to iterate through the resulting Triples. This means iterating 100 Triples will perform access control 100 times which is in most cases not necessary. This feature is only useful if during iteration the user may lose the required permissions. In many cases, it is enough to check permissions once before performing a series of actions on the TripleCollection. Therefore, SecuredMGraph provides a method called getUnsecuredMGraph() which returns the wrapped LockableMGraph if the user has all access rights. If she only has the read access right, then a WriteBlockedMGraph (write-blocked LockableMGraph) is returned. If the user has neither the read nor the write access right, then a java.security.AccessControlException is thrown.

A WriteBlockedMGraph extends WriteBlockedTripleCollection and implements LockableMGraph. It wraps a LockableMGraph passed to it through its constructor and thereby prevents the invocation

of methods of the wrapped LockableMGraph which would otherwise modify it. This prevention mechanism is implemented in the superclass WriteBlockedTripleCollection which extends java.util.AbstractCollection<Triple> and implements TripleCollection. Any attempt to invoke the "disabled" methods will throw a ReadOnlyException, which is a subclass of java.security.AccessControlException.

Note: Current support of access control is provided on TripleCollection level. This may be sufficient for some applications. However, for some other applications, a more granular access control may be needed. In this regard, access control based on rdf:type of a resource can be useful and desirable.

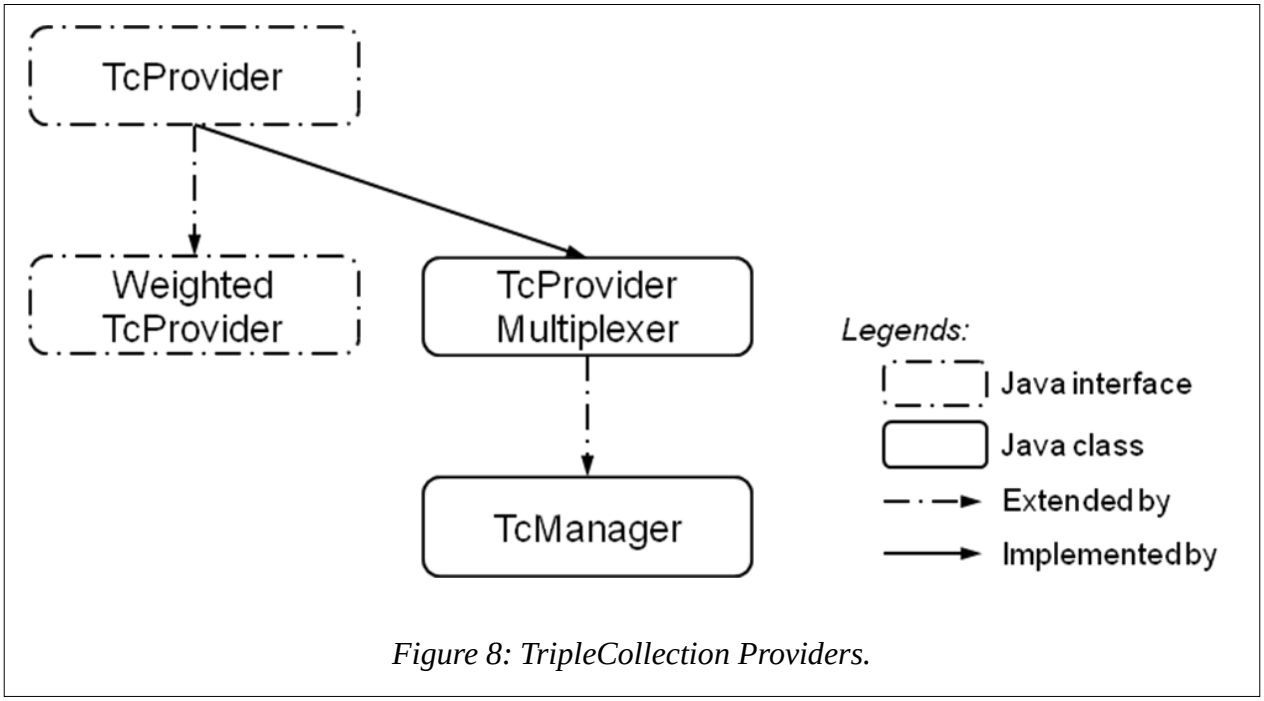

Figure 8: TripleCollection Providers.

Accessing Named TripleCollections

Giving a name to a TripleCollection is necessary if the TripleCollection should be (persistently) stored and be accessible again under the given name. Therefore, Apache Clerezza defines a so-called TcProvider (TripleCollection Provider). A TcProvider offers the functionality to create, access, and remove named TripleCollections. As depicted in Figure 8, TcProvider is an interface definition whose implementations provide public methods described in Table 4.

Table 4: Public methods of a TcProvider.

Return Value	Method Signature and Description
`Graph`	`getGraph(UriRef name)` This method returns a Graph of the specified name. It will throw a NoSuchEntityException if there is no Graph with the specified name can be found.
`MGraph`	`getMGraph(UriRef name)` This method returns an MGraph of the specified name. It will throw a NoSuchEntityException if there is no MGraph with the specified name can be found. The instances returned in different invocations are equal.
`TripleCollection`	`getTriples(UriRef name)` This method is used to get a TripleCollection of the specified name indifferently whether it's a Graph or an MGraph. It will throw a NoSuchEntityException if neither an MGraph nor a Graph can be found with the specified name.

Set<UriRef>	listGraphs() This method returns a list of the name of the Graphs available through this TcProvider. A TcProvider implementation may take into account the security context and omit Graphs which the calling thread (user) does not have the rights to access.
Set<UriRef>	listMGraphs() This method returns a list of the name of the MGraphs available through this TcProvider. A TcProvider implementation may take into account the security context and omit MGraphs which the calling thread (user) does not have the rights to access.
Set<UriRef>	listTripleCollections() This method returns a list of the name of the TripleCollections available through this TcProvider. A TcProvider implementation may take into account the security context and omit TripleCollections which the calling thread (user) does not have the rights to access.
Graph	createGraph(UriRef name, TripleCollection triples) This method creates a Graph with the specified name. The resulting Graph contains all Triples of the specified TripleCollection. However, it will throw an UnsupportedOperationException if this provider doesn't support the creation of a Graph, or an EntityAlreadyExistsException if a Graph with the specified name already exists.
MGraph	createMGraph(UriRef name) This method creates an initially empty MGraph with the specified name. However, it will throw an UnsupportedOperationException if this provider doesn't support the creation of an MGraph, or an EntityAlreadyExistsException if an MGraph with the specified name already exists.
void	deleteTripleCollection(UriRef name) This method deletes the Graph or MGraph of the specified name. If the name references a Graph which has other names, the Graph will still be available with those other names. This method will throw an UnsupportedOperationException if this provider doesn't support the deletion of a TripleCollection. It will throw a NoSuchEntityException if neither a Graph nor an MGraph with the specified name exists and it will throw an EntityUndeletableException if the TripleCollection cannot be deleted.
Set<UriRef>	getNames(Graph graph) This method returns a set of the names of the specified Graph. If the Graph is unknown, an empty set will be returned.

A WeightedTcProvider extends TcProvider with a function called getWeight() which returns a weight that can be used to prioritize available implementations (cf. Section 3.1.5). This possibility is used by TcProviderMultiplexer which has the purpose to make a set of registered WeightedTcProviders appear as one TcProvider. Thus, it implements the TcProvider interface and delegates the actual processing of a task to registered WeightedTcProviders. It attempts to satisfy a request, *e.g.*, to create an MGraph, using registered WeightedTcProviders in decreasing order of weight. If multiple WeightedTcProviders have the same weight, the lexicographical order of the fully qualified class name determines which one is used, namely the one that occurs earlier. If a WeightedTcProvider cannot fulfill this request and throws an exception, the TcProviderMultiplexer delegates the task to the next WeightedTcProvider of a lower weight. In addition to methods defined by the TcProvider interface, it provides the following two methods to register and deregister a WeightedTcProvider:


```
public void addWeightedTcProvider(WeightedTcProvider provider);
public void removeWeightedTcProvider(WeightedTcProvider provider);
```

Furthermore, an MGraph returned by TcProviderMultiplexer is ensured to be lockable, i.e. it is either a LockableMGraph or wrapped in a LockableMGraphWrapper.

Bundles providing user services which access a named TripleCollection should use a TcManager instead of a TcProviderMultiplexer, because TcManager extends TcProviderMultiplexer with access control functionality. SCB Core implements TcManager as a singleton.

OSGi Services Providing Access to Named TripleCollections

SCB Core offers different OSGi services to access a named TripleCollection. One of them is the TcManager, whose class name "org.apache.clerezza.rdf.core.access.TcManager" is used as the name for locating the TcManager service. Additionally, SCB Core provides:

- for each Graph, an OSGi service that can be located under the class name "org.apache.clerezza.rdf.core.Graph"
- for each MGraph, an OSGi service that can be located under the class names: "org.apache.clerezza.rdf.core.MGraph" and "org.apache.clerezza.rdf.core.access.LockableMGraph"

In both cases, the name of the Graph respectively MGraph is used as the value of a service property called "name" when those OSGi services are registered to the OSGi framework. This service property allows for service filtering in order to refer directly to a specific TripleCollection. The following code snippet using the package org.apache.felix.scr.annotations is an example to refer to an MGraph named urn:x-localinstance:/system.graph as an OSGi service:

```
…
public class MyClass {
    @Reference(target = "(name=urn:x-localinstance:/system.graph)")
    private MGraph systemGraph;
    …
}
```

Furthermore, instead of a TripleCollection service object, the SCB Core registers a service factory to the OSGi framework, i.e. in case of Graph, the GraphServiceFactory, and in case of MGraph, the MGraphServiceFactory is registered. Both service factories implement org.osgi.framework.ServiceFactory and have access to the TcManager instance. Registrations and deregistrations of those service factories are taken care of by the TcManager. An MGraph or a LockableMGraph instantiated by the MGraphServiceFactory is a SecuredMGraph wrapping the MGraph returned by the method getMGraph() of the TcManager, whereas a Graph instantiated by the GraphServiceFactory is a SimpleGraph constructed from a SecuredTripleCollection which wraps the Graph returned by the method getGraph() of the TcManager.

Note that WeightedTcProviders are implemented as OSGi services and the TcManager has references to them. If a WeightedTcProvider is registered to the OSGi framework, its TripleCollections are obtained and registered by the TcManager using service factories mentioned above to the OSGi framework. However, a TripleCollection will be registered only if there are no other higher weighted providers providing a TripleCollection of the same name. Moreover, an existing TripleCollection of the same name provided by a lower weighted provider will be deregistered.

In summary, SCB Core provides TcManager as an OSGi service to access TripleCollections securely, and the TcManager registers each named TripleCollection as an OSGi service to the OSGi framework. A TripleCollection as an OSGi service supports access control.

Support of SPARQL

SCB Core provides a query model which allows SPARQL [16] queries to be constructed programmatically. Figure 9 depicts Java interfaces and classes which model the four types of SPARQL queries: Ask, Select, Construct, and Describe. A SPARQL query may contain a dataset specification and a query pattern specification. A query pattern is a combination of various graph patterns specified in a WHERE clause. SimpleQuery provides a method to set this dataset and a method to set the query pattern. In SCB a dataset comprises two sets of UriRefs. One set is used to capture the Internationalized Resource Identifier (IRI) of the graph specified in each FROM clause and the other set is used to capture the IRI of the "named graph" specified in each FROM NAMED clause. For a description of the relation between RDF URI Reference and IRI, please see [8]. The FROM NAMED clause is used to specify IRI that will be referred to in a special graph pattern,

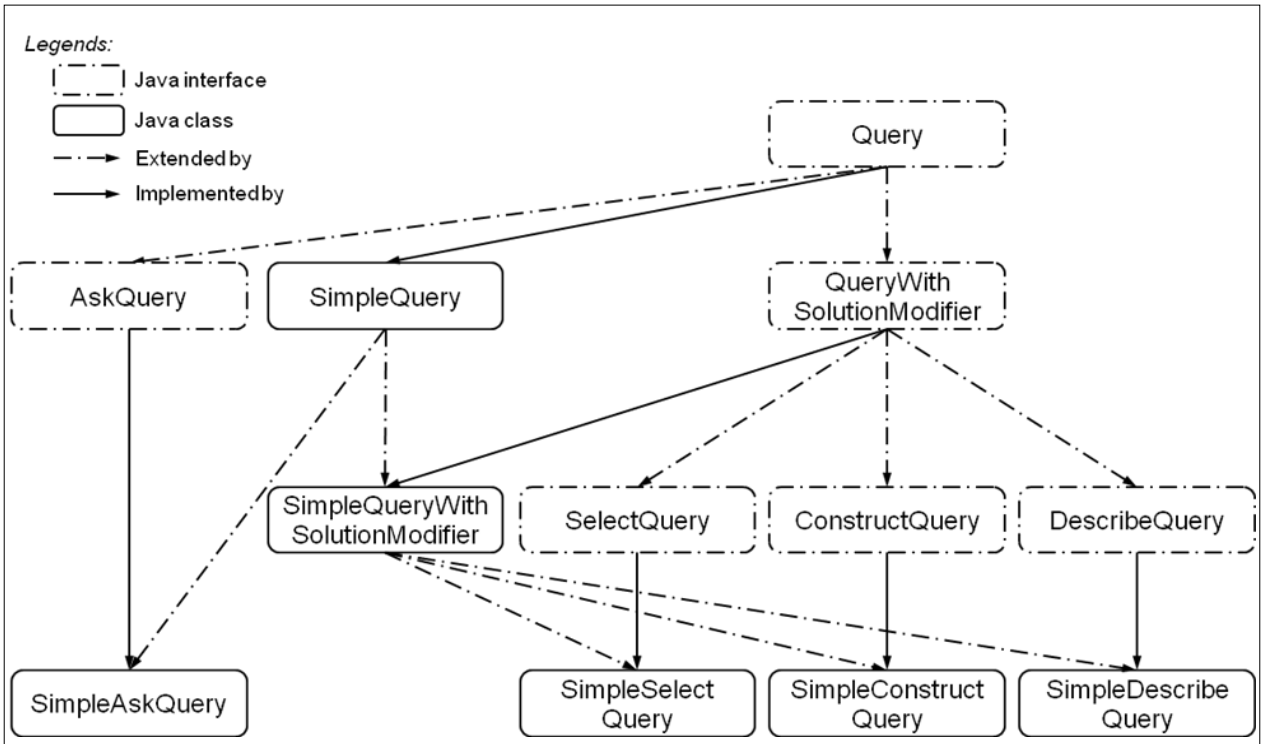

Figure 9: Modeling various types of SPARQL queries.

namely the "Pattern on Named Graph". All graphs specified in all FROM clauses are merged into a single default graph against which a query is executed, except those parts of the query that use "Pattern on Named Graph".

There are five types of graph patterns that can be combined to build a query pattern:

- Basic Graph Pattern: represented in SCB by the interface BasicGraphPattern which is implemented by the class SimpleBasicGraphPattern
- Group Graph Pattern: represented in SCB by the interface GroupGraphPattern which is implemented by the class SimpleGroupGraphPattern
- Optional Graph Pattern: represented in SCB by the interface OptionalGraphPattern which is implemented by the class SimpleOptionalGraphPattern
- Alternative Graph Pattern: represented in SCB by the interface AlternativeGraphPattern which is implemented by the class SimpleAlternativeGraphPattern

- Pattern on Named Graph: represented in SCB by the interface GraphGraphPattern which is implemented by the class SimpleGraphGraphPattern

A BasicGraphPattern contains a set of TriplePatterns, where each TriplePattern consists of a subject, a predicate, and an object. The subject and object are of type ResourceOrVariable, whereas the predicate is of type UriRefOrVariable. The class ResourceOrVariable wraps either a Resource or a Variable, while the class UriRefOrVariable wraps either a UriRef or a Variable. The query pattern in a WHERE clause is grammatically a GroupGraphPattern. A GroupGraphPattern consists of a list of graph patterns and a list of expressions as filter constraints. An OptionalGraphPattern consists of a main graph pattern and a GroupGraphPattern as an optional pattern, whereas an AlternativeGraphPattern is a list of GroupGraphPatterns. Finally, a GraphGraphPattern comprises a UriRefOrVariable to refer to the graph in each FROM NAMED clause and a GroupGraphPattern to be matched in the "named graph".

Note that at the time of writing this report, SCB Core only supports SPARQL 1.0. It has a class called SimpleStringQuerySerializer which provides a method serialize() to serialize any of the four types of SPARQL queries into a String. Furthermore, in order to parse a SPARQL query string and generate a Query object of it, SCB Core defines a class named QueryParser which implements an OSGi service and provides the method parse(). Finally, to execute a SPARQL query, a QueryEngine is needed. The QueryEngine interface is defined as follows:

```
public interface QueryEngine {
    public Object execute(TcManager tcManager, TripleCollection dafaultGraph,
            Query query);
}
```

The TcManager is passed to the method execute() to allow a QueryEngine implementation to access other TripleCollections specified in the Query. In order to ease usage of any QueryEngine, TcManager provides methods to execute each type of SPARQL query. However, it won't execute the query itself, but will delegate the execution to a QueryEngine service. Thus, in Apache Clerezza, QueryEngine implementations are expected to provide an OSGi service for the interface org.apache.clerezza.rdf.core.sparql.QueryEngine. One QueryEngine service will then be bound to the TcManager. The following methods are provided by TcManager to execute a SPARQL query:

```
public Object executeSparqlQuery(Query query,
        TripleCollection defaultGraph);
public ResultSet executeSparqlQuery(SelectQuery query,
        TripleCollection defaultGraph);
public boolean executeSparqlQuery(AskQuery query,
        TripleCollection defaultGraph);
public Graph executeSparqlQuery(DescribeQuery query,
        TripleCollection defaultGraph);
public Graph executeSparqlQuery(ConstructQuery query,
        TripleCollection defaultGraph);
```

The ResultSet returned by the execution of a SelectQuery is an Iterator over a SolutionMapping which extends Map<Variable, Resource>.

Based on various supports introduced so far, Figure 10 depicts OSGi services required by the TcManager to provide its functionality.

Support of Parsing and Serialization of TripleCollections

SCB Core also provides an OSGi service implemented by the class Parser to parse a serialized graph (TripleCollection) from an InputStream into a Graph or an MGraph, as well as an OSGi service implemented by the class Serializer to serialize a TripleCollection to an OutputStream. The Parser and Serializer are implemented as a singleton class. Various serialization formats are supported by the Parser as well as the Serializer.

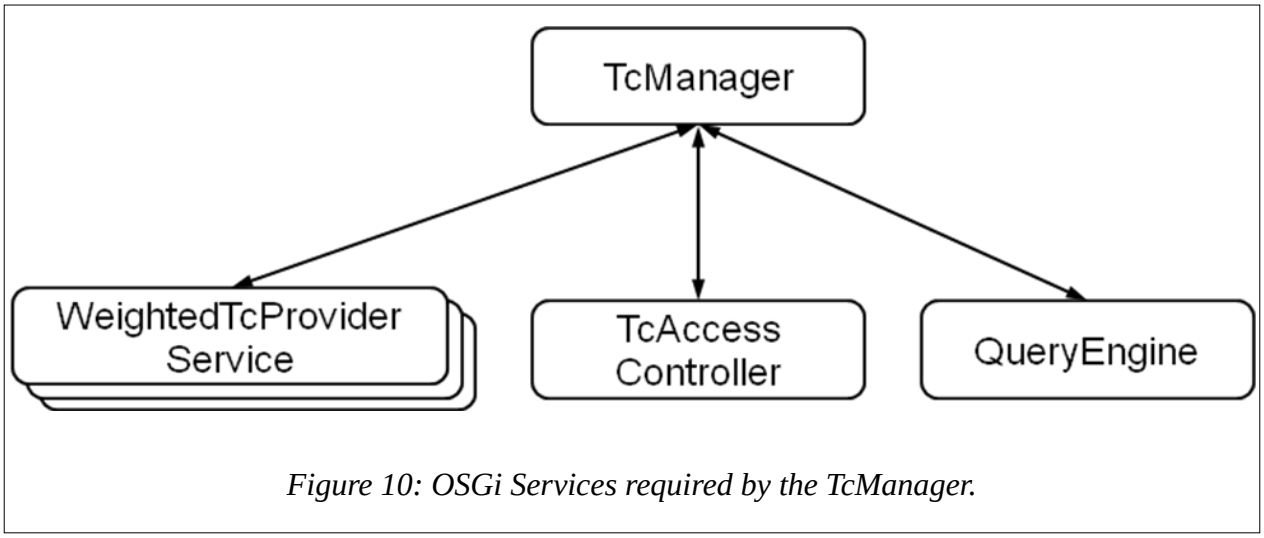

Figure 10: OSGi Services required by the TcManager.

The following methods are implemented by the Parser:

```
public Graph parse(InputStream serializedGraph,
        String formatIdentifier, UriRef baseUri);
public void parse(MGraph target, InputStream serializedGraph,
        String formatIdentifier, UriRef baseUri);
```

The baseUri is the URI against which relative UriRefs are evaluated.

The following method is supported by the Serializer:

```
public void serialize(OutputStream serializedGraph, TripleCollection tc,
        String formatIdentifier);
```

The real processing of the parsing and serialization task is done by the underlying Parsing Provider respectively Serializing Provider which are bound as OSGi services to the Parser respectively the Serializer. The corresponding interfaces are defined as follows:

```
public interface ParsingProvider {
    void parse(MGraph target, InputStream serializedGraph,
            String formatIdentifier, UriRef baseUri);
}
public interface SerializingProvider {
    public void serialize(OutputStream outputStream,
            TripleCollection tc, String formatIdentifier);
}
```

3.1.2 Ontologies

Apache Clerezza provides a set of Java classes for well-known ontologies or ontologies which are used by various bundles of the platform. Each Java class contains a set of UriRef constants for a

specific ontology, where each UriRef constant either defines a class or a property (predicate) of the ontology. For example, RDF is a Java class in package org.apache.clerezza.rdf.ontologies and RDF.type is a UriRef for http://www.w3.org/1999/02/22-rdf-syntax-ns#type. Well-known ontologies made available as Java classes are amongst others: OWL, RDF, RDFS, XSD, FOAF, DCTERMS, DC, SIOC, VCARD, EXIF, and SKOS. The namespace of each of these ontologies are listed below:

- OWL: http://www.w3.org/2002/07/owl#
- RDF: http://www.w3.org/1999/02/22-rdf-syntax-ns#
- RDFS: http://www.w3.org/2000/01/rdf-schema#
- XSD: http://www.w3.org/2001/XMLSchema#
- FOAF: http://purl.org/dc/elements/1.1/
- DCTERMS: http://purl.org/dc/terms/
- DC: http://purl.org/dc/elements/1.1/
- SIOC: http://rdfs.org/sioc/ns#
- VCARD: http://www.w3.org/2006/vcard/ns#
- EXIF: http://www.w3.org/2003/12/exif/ns#
- SKOS: http://www.w3.org/2008/05/skos#

3.1.3 SCB Utilities

To ease manipulations of TripleCollections, Apache Clerezza provides a set of utility classes: GraphNode, RdfList, UnionMGraph, and MGraphUtils. A GraphNode is an object which represents a node (RDF resource) in a TripleCollection. It provides useful methods to obtain and add information about the node. An RdfList is an implementation of java.util.List backed by an RDF collection (rdf:List). It extends java.util.AbstractList<Resource>. The list allows modifications that are reflected to the underlying TripleCollection. It reads the data from the TripleCollection when it is first needed, so changes to the TripleCollection affecting the rdf:List may or may not have an effect on the values returned by instances of this class. For that reason, only one instance of this class should be used for accessing an rdf:List or sublists thereof, when the lists are being modified. Having multiple lists exclusively for read operations (such as for immutable TripleCollections) is not problematic. The main public methods of GraphNode and RdfList are described in Table 5 and Table 6 respectively.

A UnionMGraph represents the union of multiple TripleCollections. A UnionMGraph appears like a single TripleCollection. However, current implementation of UnionMGraph does not hide duplicate triples existing in two different TripleCollections. Furthermore, write operations will only be performed on the first TripleCollection in the union which must be an MGraph. The class UnionMGraph extends AbstractMGraph and implements LockableMGraph. Finally, the class MGraphUtils provides a single method to manipulate an MGraph, namely to remove a subgraph from an MGraph.

```
public static void removeSubGraph(MGraph mGraph,
        TripleCollection subGraph) throws NoSuchSubGraphException;
public static class NoSuchSubGraphException extends Exception {};
```

The specified subgraph in removeSubGraph() must match a subgraph of the specified MGraph, so that for every node in the specified subgraph, each triple it appears in is also present in the specified MGraph. And two blank nodes are considered equals if their contexts are equals.

Table 5: Main public methods of GraphNode.

Return Value	Method Signature and Description
	`GraphNode(Resource resource, TripleCollection graph)` This constructor method returns a GraphNode containing the specified Resource as the node which is supposed to exist in the specified TripleCollection.
`TripleCollection`	`getGraph()` Returns the TripleCollection which was passed to the constructor method.
`Resource`	`getNode()` Returns the node of the GraphNode.
`Graph`	`getNodeContext()` Returns a Graph containing the context of the node. The context of a node N are the triples containing the node N as subject or object and recursively the context of all blank nodes in any of these triples. If the UriRef of the node N does not contain the character "#", its context also consists of the context of the nodes in any of those triples above whose UriRef starts with the UriRef of the node N followed by the character "#". The triples in the Graph returned by this method contain the same blank node instances as in the original TripleCollection.
`void`	`deleteNodeContext()` Deletes the context of the node from the TripleCollection.
`int`	`countObjects(UriRef predicate)` Returns the number of triples in the TripleCollection with this node as subject and the specified predicate as predicate.
`Iterator<Resource>`	`getObjects(UriRef predicate)` Returns an Iterator over all objects of triples with this node as subject and the specified predicate as predicate.
`Iterator<Literal>`	`getLiterals(UriRef predicate)` Returns an Iterator over all Literal objects of triples with this node as subject and the specified predicate as predicate.
`Iterator<UriRef>`	`getUriRefObjects(UriRef predicate)` Returns an Iterator over all UriRef objects of triples with this node as subject and the specified predicate as predicate.
`boolean`	`hasProperty(UriRef predicate, Resource object)` Checks whether this node has the given property with the given value. If the given value is null, then it is checked if this node has the specified property regardless of its value.
`int`	`countSubjects(UriRef predicate)` Returns the number of triples in the TripleCollection with this node as object and the specified predicate as predicate.
`Iterator <NonLiteral>`	`getSubjects(UriRef predicate)` Returns an Iterator over all subjects of triples with this node as object and the specified predicate as predicate.
`Iterator<UriRef>`	`getProperties()` Returns an Iterator over all properties of this node as subject.
`Iterator<UriRef>`	`getInverseProperties()` Returns an Iterator over all inverse properties of this node (this node is the object

	in the statements).
void	`addProperty(UriRef predicate, Resource object)` If the node is a NonLiteral, adds a triple with the node as subject, the specified predicate as predicate, and the specified object as object. If the node is not a NonLiteral, a RuntimeException is thrown.
void	`addPropertyValue(UriRef predicate, Object value)` If the node is not a NonLiteral, a RuntimeException is thrown. Otherwise, converts the specified value into a TypedLiteral and adds a triple with the node as subject, the specified predicate as predicate, and the resulting TypedLiteral as object.
void	`addInverseProperty(UriRef predicate, Resource subject)` If the specified subject is not a NonLiteral, a RuntimeException is thrown. Otherwise, adds a triple with the node as object, the specified predicate as predicate, and the specified subject as subject.
void	`deleteProperties(UriRef predicate)` Deletes all triples with the node as subject and the specified predicate as predicate.
void	`deleteProperty(UriRef predicate, Resource object)` Deletes a triple with the node as subject, the specified predicate as predicate, and the specified object as object.
GraphNode	`replaceWith(NonLiteral replacement)` Replaces all occurrences of the node in the TripleCollection with the specified NonLiteral, where the node is either a subject or an object in a triple.
Iterator<GraphNode>	`getObjectNodes(UriRef predicate)` Returns an Iterator over all GraphNodes, where their nodes are objects in triples with this node as the subject and the specified predicate as predicate. Those GraphNodes uses the same TripleCollection as this GraphNode.
Iterator<GraphNode>	`getSubjectNodes(UriRef predicate)` Returns an Iterator over all GraphNodes, where their nodes are subjects in triples with this node as the object and the specified predicate as predicate. Those GraphNodes uses the same TripleCollection as this GraphNode.
List<Resource>	`asList()` Creates and returns an RdfList using the node and TripleCollection of this GraphNode.
boolean	`equals(Object obj)` Returns true if obj is an instance of the same class representing the same node in the same TripleCollection. Subclasses may have different identity criteria.
Lock	`readLock()` Returns a Lock used for reading. If the TripleCollection is a LockableMGraph, it returns its Lock used for reading, otherwise it returns a fake Lock.
Lock	`writeLock()` Returns a Lock used for writing. If the TripleCollection is a LockableMGraph, it returns its Lock used for writing, otherwise it returns a fake Lock.

Table 6: Main public methods of RdfList.

Return Value	Method Signature and Description
	`RdfList(NonLiteral listResource, TripleCollection tc)` Constructs a list for the specified resource and TripleCollection.
	`RdfList(GraphNode listNode)` Constructs a list using the specified GraphNode.
`static RdfList`	`createEmptyList(NonLiteral listResource, TripleCollection tc)` This static method creates an empty RdfList by adding a triple to the specified TripleCollection with the specified NonLiteral as the subject, owl:sameAs as the predicate, and rdf:nil as the object.
`NonLiteral`	`getListResource()` Returns the node that represents the list.
`Resource`	`get(int index)` Returns the list element at the specified index position.
`int`	`size()` Returns the size of the list.
`void`	`add(int index, Resource element)` Adds the specified list element at the specified index position which must be greater than 0.
`Resource`	`remove(int index)` Removes the list element at the specified index position.
`static Set<RdfList>`	`findContainingLists(GraphNode element)` Returns a set of RdfLists, where each RdfList contains the node of the specified GraphNode. Sublists of other lists are not returned.
`static Set<GraphNode>`	`findContainingListNodes(GraphNode element)` Returns a set of GraphNodes, where the node of each GraphNode is the first node in a list containing the node of the specified GraphNode.

3.1.4 Jena Façade

In order to allow developers to write code against Jena's Graph API, but to use Apache Clerezza's TripleCollection underneath, Apache Clerezza provides an adaptor class called JenaGraph. JenaGraph extends com.hp.hpl.jena.graph.impl.GraphBase and implements com.hp.hpl.jena.graph.Graph. Its constructor takes as an argument a TripleCollection.

```
public JenaGraph(TripleCollection tc)
```

A JenaGraph can therefore be instantiated using either an MGraph or an Apache Clerezza Graph. An attempt to add or remove triples to respectively from a JenaGraph which is based on an immutable TripleCollection will result in an UnsupportedOperationException being thrown by the underlying TripleCollection. Typically, an instance of JenaGraph is passed as argument to com.hp.hpl.jena.rdf.model.ModelFactory#createModelForGraph to get a Jena Model.

3.1.5 Implemented WeightedTcProviders

There are several implementations of WeightedTcProvider available in Apache Clerezza as listed in Table 7. Some are built by wrapping existing Triple Stores and some are provided by Apache Clerezza itself. Each of these implementations is provided by a separate bundle.

Table 7: Various implementations of WeightedTcProvider.

TC Provider	Default Weight	Graph Model	Storage	Remarks
Simple	1	Clerezza	RAM	Uses SimpleMGraph and SimpleGraph to store triples.
Sesame	100	Sesame Graph	File system	The underlying storage technology is Sesame Triple Store.
Jena TDB	105	Jena Graph	File system	The underlying storage technology is Jena TDB
File	300	Serialization format	File system	Triples are stored in a file in a certain serialization format.
Literal Externalizer	500			This WeightedTcProvider will store literals of type http://www.w3.org/2001/XMLSchema#base64Binary in an external file, instead of within an MGraph. The MGraph will just store an UriRef that can be dereferenced to that file.

There is a special WeightedTcProvider named Literal Externalizer which has a very high weight so that it will be used by the TcManager if that service is activated. This special WeightedTcProvider prohibits literals of type http://www.w3.org/2001/XMLSchema#base64Binary to be stored in MGraphs managed by other WeightedTcProvider of lower weight. Instead, a special UriRef is stored there, replacing the base64Binary TypedLiteral. This special UriRef has "urn:x-litrep:" as the prefix, and the remaining part of the UriRef refers to the location of the file containing the replaced literal. Any named MGraph provided by the Literal Externalizer is an ExternalizingMGraph. An ExternalizingMGraph named N wraps the corresponding MGraph named N-externalizedliterals managed by another WeightedTcProvider. This wrapping allows the ExternalizingMGraph to exchange base64Binary TypedLiterals with special UriRefs and vice versa.

3.1.6 Implemented Parsing and Serializing Providers

The Parser and the Serializer described in Section 3.1.1 requires ParsingProvider respectively SerializingProvider services in order to work. The following ParsingProvider services have been implemented within Apache Clerezza:

- Sesame-based ParsingProvider
- Jena-based ParsingProvider
- RDFa ParsingProvider for parsing HTML or XHTML documents containing RDF statements (triples) specified in RDFa (Resource Description Framework – in – attributes) [1].
- RDF-JSON ParsingProvider for parsing graphs in application/rdf+json format.

Correspondingly, the following SerializingProvider services have been implemented within Apache Clerezza:

- Sesame-based SerializingProvider
- Jena-based SerializingProvider
- RDF-JSON SerializingProvider
- Stable SerializingProvider

The Stable SerializingProvider is a special SerializingProvider that tries to provide similar results when serializing TripleCollections. Specifically, it tries to label blank nodes deterministically with reasonable complexity. It only supports the text/rdf+nt format and does not guarantee a deterministic result, but it may minimize the amount of modified lines in serialized output. The algorithm for labeling blank nodes is based on the work by Jeremy J. Carroll [3]. The Stable SerializingProvider also uses the algorithm for Minimum Self-contained Graph (MSG) decomposition developed by Giovanni Tummarello et al. [14].

3.1.7 Jena-based SPARQL Engine

Apache Clerezza provides an implementation of the QueryEngine interface (cf. Section 3.1.1) based on the Jena SPARQL support. The class JenaSparqlEngine implements that interface as an OSGi service which will be used by the TcManager.

3.1.8 Composite Resource Indexing Service (CRIS)

CRIS is based on Apache Lucene [18] and provides means to index RDF resources. It works by indexing the values of properties on a resource. This enables to search for the property values using CRIS. The results that CRIS delivers are the corresponding RDF resources.

GraphIndexer

The core of CRIS is the GraphIndexer class. Note that GraphIndexer is not an OSGi service, but it has to be instantiated by the user to provide an index. The GraphIndexer needs two graphs to work with. One graph contains the IndexDefinitions, that is the specification of which resources and properties to index (see IndexDefinitionManager). The other graph is the the graph that contains the resources to index. Note that CRIS indexes RDF resources based on their rdf:type and that the indexing works on a per-property basis. That means, not all properties on a resource are indexed by default. The user has to specify which properties to index.

GraphIndexer also provides the interface to search for resources using the findResources() method. The search is specified using Conditions and optionally a SortSpecification and FacetCollectors. The findResources() method is overloaded with methods that allow the specification of the resource type and search query directly.

IndexDefinitionManager

The IndexDefinitionManager helps to manage indexing specifications using the CRIS ontology in the index definition graph (see GraphIndexer). Indexing is enabled for resources according to their rdf:type. Additionally, the index definitions specify the properties of the resource that are indexed. One can think of an index definition as specifying the keys (properties) that are mapped to the value (the resource URI) in the index.

VirtualProperty

VirtualProperties represent RDF properties but they add functionality required to index them. There are several types of VirtualProperties that can be used for specific indexing requirements:

- PropertyHolder: This is a VirtualProperty-adapter for a single RDF property.
- JoinVirtualProperty: This creates a virtual property which represents several RDF properties and the object is a literal that concatenates the objects of the specified RDF properties into a single literal separated by a space (*e.g.*, this can represent a concatenation of foaf:firstName and foaf:lastName. If the first name is “John” and the last name is “Doe”, the value of the JoinVirtualProperty will be “John Doe”. Note that the order of concatenation matters. If it is switched, the value will be “Doe John”).
- PathVirtualProperty: This can be used to index properties that are not attached directly to the

indexed resource. For example, one can index a property value that is attached to a blank node, if the blank node is in turn attached to the indexed resource.

Conditions

Conditions represent the type of query to perform on the index. When searching for resources, a list of conditions can be supplied. These conditions are applied using a boolean and relationship between them. Currently, there are the following Conditions implemented:

- WildcardCondition: Takes a string query as input. The query can contain wild-card characters ("*" for any number of characters, "?" for a single character). The WildcardCondition has to be specified for a single property. When querying multiple properties, multiple WildcardConditions have to be used. Note that the property has to be indexed.
- TermRangeCondition: Returns results that lie between (according to String.compareTo) a lower and an upper term specified by the user. The TermRangeCondition has to be specified for a single property. When querying multiple properties, multiple TermRangeCondition have to be used. Note that the property has to be indexed.
- GenericCondition: This condition supports full Lucene query parser syntax [19]. It supports querying on multiple properties directly. Note that all supplied properties have to be indexed.

SortSpecification (Sorting Search Results)

A SortSpecification contains an ordered list of VirtualProperties. The order of addition defines the significance for sorting (all search results are sorted according to the first property, then according to the second property, etc). Note that the added properties have to be indexed.

Example:

Unsorted results: {myuri#adam_berkley, myuri#berta_adams}

SortSpecification: {foaf:lastName, foaf:firstName}

Sorted results: {myuri#berta_adams, myuri#adam_berkley}

In order to specify sorting by indexing order (the order in which the resources have been added to the index) or according to relevance (document score computed by Lucene), there are two special objects: SortSpecification.INDEX_ORDER and SortSpecification.RELEVANCE. These objects can be added to the SortSpecification like VirtualProperties. A usage example could be to use INDEX_ORDER as the last entry in the list to break ties in a well specified manner. If two resources are sorted equally, it is undefined which resource is displayed first. When the final step is to order by INDEX_ORDER, then the order in this case is defined by the time the resource in question has been indexed and it will be guaranteed to be the same each time.

Besides the order of properties, the user has to specify how to interpret the value of the property in order to receive expected results. The supported value types are defined as static constants on SortSpecification. The most commonly used value type is STRING. This type interprets the literal values as a String for sorting and returns results according to their "natural order". There is a STRING_COMPARETO constant as well but this is much more resource intensive. It uses the String.compareTo() method for sorting. Only use this in case STRING does not return expected results. Other useful types are INT and FLOAT.

By default the sorting is in ascending order. When properties are added it to the SortSpecification it is possible to specify that the order should be reverse (descending).

FacetCollector (Faceted Search)

A FacetCollector can be supplied to perform a faceted search. It works on a per-property basis. The

user can add properties for which the FacetCollectors collects facets. That means, it groups certain information according to the values of the specified property. The information collected per facet is depending on the FacetCollector implementation.

An example of facets is:

Property to collect facets for: foaf:firstName

Search Results: {myuri#adam_berkley, myuri#berta_adams, myuri#adam_hawk}

Facets: {“adam”, “berta”}

A FacetCollector can collect facets for multiple properties. Facets are available as sets of entries. The entry has a key (*e.g.*, “berta”) and a value (the information collected for the berta facet).

Currently two basic FacetCollectors are implemented:

- CountFacetCollector: Counts the number of occurrences of a facet as the information associated to it (for the example above: {“adam”, 2}, {“berta”, 1}).
- SortedCountFacetCollector: The same as CountFacetCollector but it returns the facets ordered by value.

Example Usage

```
MGraph definitions = new SimpleMGraph(); //indexing specifications
MGraph dataGraph = new SimpleMGraph(); //the graph to index

//adding index definitions:
IndexDefinitionManager indexMgr = new IndexDefinitionManager(definitions);

// index firstName + " " + lastName;
List<VirtualProperty> predicates = new ArrayList<VirtualProperty>();
predicates.add(new PropertyHolder(FOAF.firstName));
predicates.add(new PropertyHolder(FOAF.lastName));
JoinVirtualProperty name = new JoinVirtualProperty(predicates);

// index the value on the path res/vcard:adr/vcard:street:address
List<VirtualProperty> path = new ArrayList<VirtualProperty>();
path.add(VCARD.adr);
path.add(VCARD.street_address);
PathVirtualProperty streetAddress = new PathVirtualProperty(path);

List<VirtualProperty> properties = new ArrayList<VirtualProperty>();
properties.add(new PropertyHolder(FOAF.mbox)); //index the value of foaf:mbox
properties.add(name);
properties.add(streetAddress);

indexMgr.addDefinition(FOAF.Person, properties); //index resources with rdf:type
foaf:Person

//create index
```



```
GraphIndexer service = new GraphIndexer(definitions, dataGraph); //creates the
index in memory


//add more data to dataGraph if necessary


//sort specification
SortSpecification sortSpecification = new SortSpecification();
sortSpecification.add(name, SortSpecification.STRING); //sort by name


//faceted search
SortedCountFacetCollector facetCollector = new SortedCountFacetCollector();
//count occurrence of same value
facetCollector.addFacetProperty(streetAddress); //count street address values


//search for name
List<NonLiteral> results = service.findResources(name, "John D*", false,
sortSpecification, facetCollector);
//results contains resources with names such as "John Dalton", "John Doe", etc.
sorted by name


facetCollector.getFacets(streetAddress); //contains counts of how often the same
streetAddress occurs
service.closeLuceneIndex(); //release reources
```

3.1.9 Web Interface

A set of functionality to access and manipulate TripleCollections is provided by Apache Clerezza through Web services as listed and described in Table 8.

Table 8: Web services to access and manipulate TripleCollections.

URL Path	Service Description
/sparql	This service allows for querying TripleCollections using GET or POST. It returns either a Graph or a javax.xml.transform.dom.DOMSource. A Graph is returned if a CONSTRUCT or a DESCRIBE query was submitted and successfully carried out. A DOMSource is returned if an ASK or a SELECT query was submitted and successfully carried out. The result can be transformed using XSL. The POST method accepts the following form parameters: *query* (the query string), *default-graph-uri*, *apply-style-sheet*, *server-side*, and *style-sheet-uri*. The value of apply-style-sheet is either "on" or "off" which denotes whether an XSL stylesheet should be applied or not. The value of server-side is either "on" or "off" which denotes whether the XSL transformation should be performed on the server side or not. The GET method accepts the following query parameters: *query*, *default-graph-uri*, *server-side*, and *style-sheet-uri*. If style-sheet-uri is not null, then application of the XSL stylesheet is implied. The value of server-side is either "true" or "false" which denotes whether the XSL transformation should be performed on the server side or not.

/graph	This service allows for retrieval of a named TripleCollection using GET and for replacing all triples of or for adding triples to a named MGraph with the specified triples using POST. The GET method accepts a single query parameter *name* which refers to the name of the TripleCollection. The response contains the requested triples of the specified TripleCollection. The POST method accepts a multipart/form-data which consists of: • a file labeled "graph" containing the triples and the mime-type • a text field labeled "name" specifying the name of the MGraph • an optional text field labeled "mode" specifying the mode. If the mode is "replace", existing triples of the MGraph will be deleted before new triples are added. If the mode is not specified or is "append", posted triples are added to the MGraph • an optional text field labeled "redirection" specifying an URI which the client should be redirected to in case of success. The POST method returns a response with status code: • BAD REQUEST (400) if the required data are missing. • SEE OTHER (303), if redirection is specified. • CREATED (201), if redirection is not specified and a new MGraph is created. • NO CONTENT (204), if redirection is not specified and no new MGraph is created.
/admin/backup/download	This service allows for retrieval of a zip file containing all TripleCollections that the user has access to. The TripleCollections are serialized in N-Triples format.
/admin/graphs/smush	This service allows for removal of duplicate nodes in an MGraph using POST. The POST method accepts a form parameter called *graphName*. Currently, only nodes with a shared inverse functional property are considered the same. For example, if foaf:mbox is an inverse functional property, then both blank nodes _:b1 and _:b2 are the same if they have the property foaf:mbox pointing to the same object node.

3.2 Triaxrs

Triaxrs is an implementation of the JAX-RS specification 1.0. The design of Triaxrs has been described in detail in [6]. The implementation supports operations in an OSGi runtime environment. However, not all functions specified by JAX-RS 1.0 are implemented. For example, the class RuntimeDelegateImpl, which is the current implementation of a concrete subclass of RuntimeDelegate, does not support the creation of UriBuilder and VariantListBuilder. Nevertheless, main functions and all annotations are supported to meet the requirements of most JAX-RS based Web applications. Additionally, Triaxrs introduces some extensions which are useful in some use cases.

3.2.1 Implemented Entity Providers

As described in [6], entity providers supply mapping services between representations and their associated Java classes. There are 2 types of Entity Providers: Message Body Reader (MBR) and Message Body Writer (MBW). Table 9 and Table 10 list Message Body Writers respectively Message Body Readers implemented by Apache Clerezza.

By default the GraphNode MBW produces a serialization of the context of the node represented by the GraphNode. This can be expanded by using the query parameters xPropObj and xPropSubj.

These parameters specify predicates whose objects respectively subjects are expanded as if they were blank nodes.

Table 9: Message Body Writers.

Represented Java Class	Produced Media Type
String	text/plain, */*
byte[]	*/*
java.io.File	*/*
javax.ws.rs.core.MultivaluedMap<String, ? extends Object> (defined by JAX-RS for HTML form's content type application/x-www-form-urlencoded)	application/x-www-form-urlencoded
java.io.InputStream	*/*, application/octet-stream
java.io.Reader	*/*
javax.xml.transform.Source	text/xml, application/xml, */*
javax.ws.rs.core.StreamingOutput	*/*
javax.activation.DataSource	*/*
org.json.simple.JSONObject	application/json
org.apache.clerezza.rdf.core.TripleCollection	text/rdf+n3, text/rdf+nt, application/rdf+xml, text/turtle, application/x-turtle, application/rdf+json
org.apache.clerezza.rdf.utils.GraphNode	text/rdf+n3, text/rdf+nt, application/rdf+xml, text/turtle, application/x-turtle, application/rdf+json

Table 10: Message Body Readers.

Represented Java Class	Consumed Media Type
String	text/plain, */*
byte[]	*/*
java.io.File	*/*
javax.ws.rs.core.MultivaluedMap<String, String> (defined by JAX-RS for HTML form's content type application/x-www-form-urlencoded)	application/x-www-form-urlencoded
java.io.InputStream	*/*, application/octet-stream
java.io.Reader	*/*
javax.xml.transform.stream.StreamSource	text/xml, application/xml, */*
javax.xml.transform.sax.SAXSource	text/xml, application/xml, */*
javax.xml.transform.dom.DOMSource	text/xml, application/xml, */*
javax.activation.DataSource	*/*
org.apache.clerezza.jaxrs.utils.form.MultiPartBody	multipart/form-data
org.apache.clerezza.rdf.core.TripleCollection	text/rdf+n3, text/rdf+nt, application/rdf+xml, text/turtle, application/x-turtle, application/rdf+json

org.apache.clerezza.rdf.core.Graph	text/rdf+n3, text/rdf+nt, application/rdf+xml, text/turtle, application/x-turtle, application/rdf+json

3.2.2 Supported Context Types

Currently, Triaxrs does not implement ContextResolver specified in JAX-RS, but it supports the injections of information into class fields and method parameters using @Context annotation for the following Types defined by JAX-RS: UriInfo, Request, HttpHeaders, and Providers. However, the SecurityContext Type is not supported yet.

3.2.3 Bundle Path Prefix

Triaxrs allows bundles providing JAX-RS resources to specify a path prefix to be prepended to the path of the JAX-RS root resource class. This prefix can be specified in the MANIFEST header named Triaxrs-PathPrefix. The file (called MANIFEST.MF) containing this information is located in the META-INF folder of the OSGi bundle. For example, suppose a bundle is configured to set the Triaxrs-PathPrefix to "foo" and the host running the Apache Clerezza instance is named example.org. Now, if a JAX-RS root resource class in this bundle is annotated with @Path("test") and there is a resource method defined for HTTP GET in this class, then the Web service provided by this resource method can be access through the URL: http://example.org/foo/test instead of http://example.org/test.

Furthermore, the support of bundle path prefix becomes handy if there are more than one bundle implementing a JAX-RS provider which would match the same criteria, *e.g.*, in case of Message Body Writers, they have the same produced media types and are applicable to the same Java class. In this respect, the bundle path prefix can be used to select the more appropriate JAX-RS provider by matching the bundle path prefix with the request URI.

3.3 Static Web

Figure 11 depicts bundles of the Static Web group which are briefly described in Table 11. In this

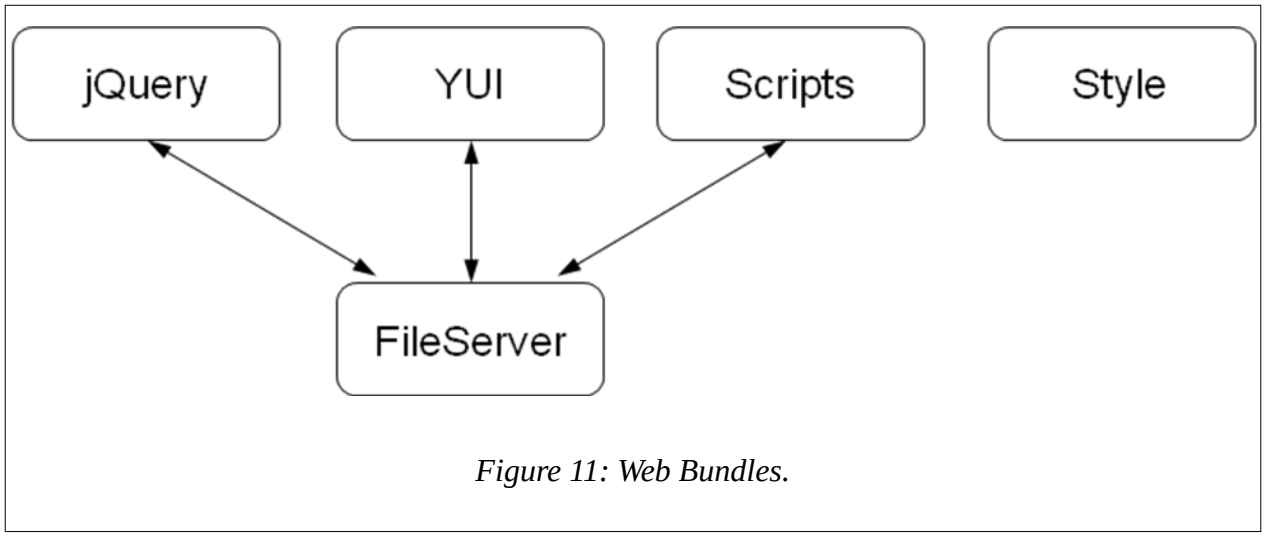

Figure 11: Web Bundles.

table, the column named "Starting Path" specifies the starting path of the URL to access the static Web resources provided by the respective bundle.

Table 11: Static Web Bundles

Bundles	Starting Path	Remarks
jQuery	/jquery/	jQuery is best summarized with a citation taken from its Website: "jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development"
YUI	/yui/2/ /yui/3/	YUI (Yahoo! User Interface) is a JavaScript and CSS Library developed by Yahoo!. Version 2 and 3 of YUI is made available to clients through this bundle.
Scripts	/scripts/	This bundle provides access to a collection of JavaScripts.
FileServer	/	This bundle provides a mechanism to access static Web resources.
Style	/style	This bundle provides access to a collection of CSS files.

3.4 Platform

This section describes functionality supported by Platform bundles.

3.4.1 Information Resource Modeling

Apache Clerezza supports an information resource modeling as described on the Website http://discobits.org/. Reto Bachmann-Gmür, the creator of the site, introduces the terms InfoBit and DiscoBit ('Disco' stands for 'Discourse') as follows: "*Using RDF models often involves associating things of the described world to data that can be shown - or made perceivable by other means - to a user. By doing so an entity of the described domain (e.g. a foaf:Person) is linked to a describing node, the latter would typically be a literal or a dereferenceable URI. While using dereferenceable URIs keeps the describing entities ouside the model it offers the advantage of the protocol used to get the represenation. Using content and language negotiatiation the represantation will be optimized for the abilities of the device and of the user. The classes d:InfoBit and d:DiscoBit are the main classes of the ontology, the prefix "d:" refers to the namespace http://discobits.org/ontology#.*"

The InfoBit and DiscoBit are further defined and described on the site as follows: "*Discobits allows to bring describing entities into the model an express relationships similar to multipart-mime message (RFC 2045). It can express that an entity contains different other entities and about the role an entity plays with a containing entity. Such a describing entity is a DiscoBit. A DiscoBit is not necessarily a n entity of description it may also be a novel, a movie or something else that's rather creating than describing. The InfoBit is something that can be made perceivable by a computer. For example the interpretation as image/gif of some bytes is an InfoBit. Any RDFS Literal is an InfoBit. An InfoBit has no content or meaning beyond the mere set of possible representations to a user. InfoBits are the lowest common denominators of the modeled world, an image, a sound, a text, a program.*"

The relation between InfoBits and DiscoBits is illustrated by the following excerpt: "*When humans say things about InfoBits that weren't inferable beforehand to the system, they create DiscoBits ('Disco' stands for 'Discourse'). A file in a directory is a DiscoBit by creating a functional role associated to a specific InfoBit. A DiscoBit either refers to one InfoBit, or to a set entries (d:Entry) which refer to other DiscoBits. These entries describe either parts or variants of the containing DiscoBit.*"

The ontology http://discobits.org/ontology# is supposed to be used for modeling information as it is found on the Web or in emails. The main classes are DiscoBit and InfoBit. An InfoBit is something that can be shown by a computer, whereas a DiscoBit is a subject of conversation related to certain

information. Whenever something is said about an InfoBit which is not implied by its nature, a DicoBit is created. A DiscoBit is the functional role of an InfoBit in a discoursive context. It is either associated to exactly one InfoBit or to a sequence of other DiscoBits. Subclasses of DiscoBit define how this sequence is to be interpreted. Figure 12 depicts subclasses of DiscoBit which are briefly defined in Table 12.

Table 12: Subclasses of DiscoBit.

DiscoBit	Description
OrderedContent	A DiscoBit whose representation is created by sequentially concatenating the InfoBits of a sequence of DiscoBits.
TitledContent	An OrderedContent with exactly two elements, where the first contained DiscoBit is a title.
InfoDiscoBit	A DiscoBit associated to exactly one InfoBit.
XHTMLInfoDiscoBit	A DiscoBit associated to exactly one InfoBit of type XHTML.
ImageInfoDiscoBit	A DiscoBit associated to exactly one InfoBit containing an Image.
BinaryInfoDiscoBit	A DiscoBit associated to exactly one InfoBit containing binary data.

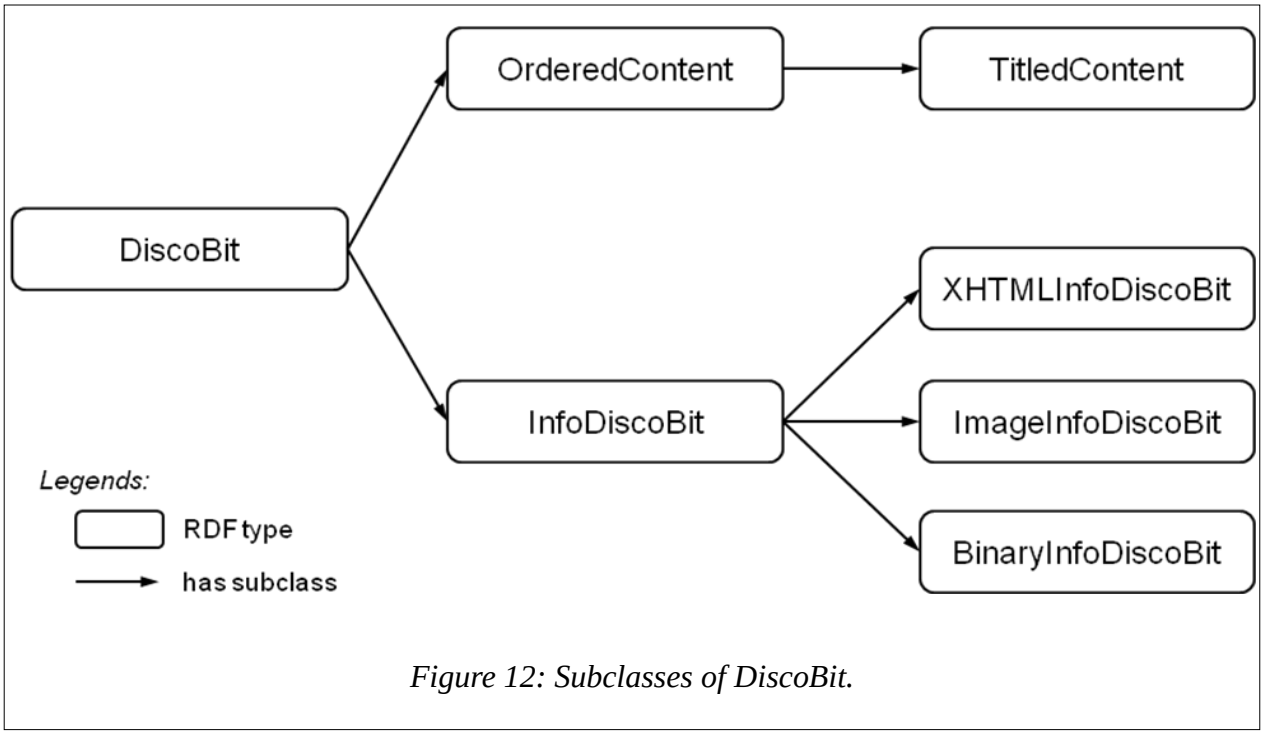

Figure 12: Subclasses of DiscoBit.

A resource of rdf:type InfoDiscoBit can have the following properties:

- media type (http://discobits.org/ontology#mediaType)
- info bit (http://discobits.org/ontology#infoBit), which points to an InfoBit
- … (others like DC.created, etc.)

A resource of rdf:type OrderedContent can have the following properties:

- contains (http://discobits.org/ontology#contains)

The range of the property "contains" must be a resource of rdf:type http://discobits.org/ontology#Entry which has the following two properties:

- holds (http://discobits.org/ontology#holds), which points to a DiscoBit
- pos (http://discobits.org/ontology#pos), which points to a http://www.w3.org/2001/XMLSchema#nonNegativeInteger denoting the position of an entry within its containing OrderedContent.

3.4.2 Platform TripleCollections

Apache Clerezza provides a set of named TripleCollections that serves various purposes as described in Table 13.

Table 13: Platform TripleCollections

TripleCollection		Type	Designated Use
Short Name	**UriRef**		
System Graph	urn:x-localinstance:/system.graph	MGraph	Stores data needed for a correct working of Apache Clerezza instance
Config Graph	urn:x-localinstance:/config.graph	MGraph	Stores data needed for configuring Apache Clerezza instance
Content Graph	urn:x-localinstance:/content.graph	MGraph	Stores application data

3.4.3 Extended Content Graph and ContentGraphProvider Service

An Extended Content Graph is a UnionMGraph of Content Graph with other TripleCollections that can be dynamically added and removed. An OSGi service that provides this Extended Content Graph is the ContentGraphProvider service. The following method of the service is used to get the Extended Content Graph:

```
public LockableMGraph getContentGraph();
```

This service has a service property called additions which can be used to configure the ContentGraphProvider service to "permanently" add the specified TripleCollections to the Extended Content Graph. Permanently means that this configuration of additional TripleCollections is not gone if the bundle (org.apache.clerezza.platform.graphprovider.content) containing the ContentGraphProvider service is restarted.

Additionally, the ContentGraphProvider service provides the following methods to temporarily add respectively remove a TripleCollections into/from the Extended Content Graph.

```
public void addTemporaryAdditionGraph(UriRef graphName);
public void removeTemporaryAdditionGraph(UriRef graphName);
```

3.4.4 GraphNodeProvider Service

This service returns a GraphNode for a specified resource. The returned GraphNode has as base graph the Extended Content Graph provided by the ContentGraphProvider service and

- for remote URIs, the Graphs they dereference to
- for local URIs with a path-section starting with /user/{username}/, the local-public-graph of that user.

All non http* URIs are assumed to be local. An URI is also local if it has the same Base-URI as the platform (org.apache.clerezza.platform.config.PlatformConfig#getBaseUris() returns a set of Base-URIs assigned to the platform. The get() method of the GraphNodeProvider class which

implements this service is defined as follows:

```
public GraphNode get(UriRef uriRef);
```

3.4.5 PageNotFoundService

To allow applications to customize the Web response with 404 (Not Found) status code, an application is expected to offer an OSGi service for the interface org.apache.clerezza.platform.content.PageNotFoundService. The interface is defined as follows:

```
public interface PageNotFoundService {

    public Response createResponse(UriInfo uriInfo);

}
```

3.4.6 Type Handling

Apache Clerezza defines a mechanism called Type Handling to process a Web request, if based on the matching algorithm defined by the JAX-RS specification, *actually*, no JAX-RS resource can be found to process the Web request. In order to allow Triaxrs to do find a JAX-RS resource that can process the Web request, Apache Clerezza provides a JAX-RS root resource class named TypeHandlerSpace which is annotated with @Path("/") and which has a sub-resource locator annotated with @Path("{path:.*}"). This way, this sub-resource locator of the TypeHandlerSpace will be selected by Triaxrs if no other more specific resource method can be found to process a Web request. In this section, the term TypeHandlerSpace also means "the abovementioned sub-resource locator annotated with @Path("{path:.*}") of the TypeHandlerSpace".

TypeHandlerSpace implements the Type Handling mechanism. Type Handling uses the rdf:type values of a resource to select a TypeHandler to process a Web request. The resource itself must be identifiable through the request URI. A TypeHandler is thus a JAX-RS sub-resource. When handling a request, TypeHandlerSpace checks if a resource with the requested URI exists in the Content Graph. Access to the Content Graph is done via the TcManager service. If a resource is found, then it uses the TypeHandlerDiscovery service to find a suitable TypeHandler to process the request. However, there is a convention defined by Apache Clerezza, that if a GET request URI ends with "-description", e.g., http://example.org/foo-description, and http://example.org/foo is a resource in the Content Graph, then TypeHandlerSpace should just return the context of the resource http://example.org/foo.

TypeHandler

The TypeHandlerDiscovery service manages a set of registered TypeHandler services. A class that offers a TypeHandler service must be a JAX-RS sub-resource which

- implements an OSGi service that can be located under the class name java.lang.Object
- has a service property org.apache.clerezza.platform.typehandler set to true
- annotated with @SupportedTypes declaring the supported rdf:type values and whether this TypeHandler is to be prepended or appended to the list of TypeHandlers. Normally, a TypeHandler is to be prepended, except when one is installing a fallback-handler.

A typical example for a class definition of a TypeHandler is as follows:

```
@Component
@Service(Object.class)
@Property(name = "org.apache.clerezza.platform.typehandler", boolValue = true)
@SupportedTypes(types = { "http://clerezza.org/2009/09/hierarchy#Collection" },
    prioritize = true)
```



```
public class CollectionTypeHandler {
    …
    @GET
    public GraphNode getResource(@Context UriInfo uriInfo) {
        …
    }
    …
}
```

RDF Type Prioritization

A resource can have several rdf:type values assigned. To determine the most appropriate TypeHandler for processing the Web request respectively the identified resource, the rdf:type values of a resource must be prioritized. Such prioritization is provided by a TypePrioritizer service. Apache Clerezza allows applications to define the prioritization of rdf:type values. This information is stored as an rdf:List in the System Graph which will be consulted by the TypePrioritizer service to sort a Collection of rdf:type values.

DiscobitsTypeHandler

If no TypeHandler can be found for the identified rdf:type values of a resource, a fallback TypeHandler is used. Apache Clerezza provides this fallback TypeHandler called DiscobitsTypeHandler which supports the rdf:type rdfs:Resource (http://www.w3.org/2000/01/rdf-schema#Resource). The sub-resource methods of the DiscobitsTypeHandler for processing GET, PUT, and DELETE requests are described in Table 14.

Table 14: Sub-resource methods of DiscobitsTypeHandler for GET, PUT, and DELETE requests.

Return Value	Method Signature and Description

Object	`@GET`
	`@Produces({"*/*"})`
	`getResource(@Context UriInfo uriInfo)`
	The resource is identified through the injected context UriInfo.
	This method returns one of the following:
	• A 404 response created through the PageNotFoundService if the resource cannot be found through the GraphNodeProvider service.
	• A 200 response with its entity body containing an InfoDiscobit if the resource can be found through the GraphNodeProvider service and the resource is of rdf:type DISCOBITS.InfoDiscoBit.
	• A GraphNode returned by the GraphNodeProvider service if the resource is not of rdf:type DISCOBITS.InfoDiscoBit.

Response	`@PUT`
	`putInfoDiscobit(@Context UriInfo uriInfo,`
	`@Context HttpHeaders headers, byte[] data)`
	Creates an InfoDiscobit in the Content Graph for the specified URI using the specified data. The resource is of rdf:type InfoDiscoBit. Its infoBit property is a base64Binary TypedLiteral of the specified byte array. The value for its mediaType property is either obtained from the Content-Type request header or guessed from the request URI.

Response	`@DELETE`
	`delete(@Context UriInfo uriInfo)`
	Deletes a resource identified by the specified URI. If the resource cannot be found in the Content Graph, a response with the status code 404 (“Not Found”)

	is returned. This method returns a response with the status code 200 (“OK”) if the request can be successfully processed. In this case, the resource node and its context are deleted from the Content Graph.

Note that, in addition to the HTTP methods listed in Table 14, the DiscobitsTypeHandler also supports HTTP methods defined by WEBDAV (HTTP Extensions for Distributed Authoring): MKCOL, COPY, MOVE, PROPFIND, PROPPATCH, LOCK, and UNLOCK.

3.4.7 Type Rendering

Message Body Writers (MBWs) offer a nice mechanism compared with coding the response to a Web request directly in a JAX-RS resource or sub-resource method. As described in [6], MBWs are intended for representing Java objects according to the supported media types. However, generating a complex response of media type “application/xhtml+xml” by writing an MBW in Java is surely not a convenient method. In this respect, the Scala programming language [20] offers a much more convenient way of producing XML documents.

Since developers using Apache Clerezza deal with rendering of resources which are nodes in RDF graphs, it would be helpful for developers if the mapping of Java classes to representations can be extended to work with GraphNodes. Instead of finding applicable MBWs to represent a Java object given its Java class and acceptable media types, applicable Renderers should be found to represent a GraphNode given the “types” of its node, a rendering mode pattern, and acceptable media types. The “types” of a node (resource) in a GraphNode is the data type of the node if it is a TypedLiteral. If the node is a NonLiteral, its “types” are the values of its rdf:type properties. A Renderer generates a representation of a resource described in TripleCollections. The rendering mode enables variations in representations to be generated for the same resource. The mechanism of finding and selecting a Renderer to render a node of a GraphNode based on the “types” of the node, and using it in the rendering process is called Type Rendering.

GenericGraphNodeMBW

An MBW is already available for GraphNodes to represent the context of the node in a GraphNode (cf. Section 3.2.1). This MBW supports typical RDF serialization formats. It does not produce “application/xhtml+xml” content. Therefore, Apache Clerezza provides a second GraphNode MBW called GenericGraphNodeMBW which should produce “application/xhtml+xml” content. It is generic in the sense that it is applicable to all “types” of nodes in a GraphNode.

The GenericGraphNodeMBW is annotated with @Produces({"application/xhtml+xml", "*/*"}) and uses the RendererFactory service to obtain a Renderer and a set of UserContextProvider services to get the user context. Then it delegates the rendering of the response node, i.e. the GraphNode, to the render() method of the Renderer as depicted in the following pseudo code:

```
Renderer renderer = rendererFactoryService.createRenderer(
    GraphNode to be rendered, the requested rendering mode,
    a list of acceptable media types);
GraphNode userContext = new GraphNode(new BNode(), new SimpleMGraph());
foreach userContextProviderService do
    userContext = userContextProviderService.addUserContext(userContext);
endfor
renderer.render(GraphNode to be rendered, userContext,
    the requested rendering mode,
    UriInfo obtained via JAX-RS injection,
```



```
    HttpHeaders obtained via JAX-RS injection,
    MultivaluedMap<String, Object> containing response headers passed to the
        GenericGraphNodeMBW by JAX-RS runtime,
    Stream to write the rendering result to);
```

UserContextProvider

A UserContextProvider service provides the following method to add user context information to the specified GraphNode.

```
public GraphNode addUserContext(GraphNode node);
```

The added information should be specifically for the currently logged in user. The method returns a GraphNode containing user context information in addition to the information already existing in the provided GraphNode. The information previously existing in the provided GraphNode is not changed by this method. The method may add the context information directly to the provided GraphNode or create a new GraphNode instance, in which case the returned GraphNode must be modifiable. In both cases, the resulting GraphNode is returned by this method.

Renderer

Apache Clerezza defines the interface Renderer whose implementations should provide the following methods:

```
public MediaType getMediaType();
public void render(GraphNode node, GraphNode userContext,
        String mode, UriInfo uriInfo,
        HttpHeaders requestHeaders,
        MultivaluedMap<String, Object> responseHeaders,
        OutputStream entityStream) throws IOException;
```

The getMediaType() method returns the media type of the rendered response. The render() method renders the specified node, possibly considering the specified user context, to the specified entityStream.

RendererFactory

A RendererFactory provides the following method to obtain the best suitable Renderer for a given set of parameters:

```
public Renderer createRenderer(GraphNode resource, String mode,
        List<MediaType> acceptableMediaTypes);
```

This method “creates” a Renderer for the specified rendering mode, acceptable media types, and the “types” of the node of the specified GraphNode. The acceptableMediaTypes list represents the media types that are acceptable for the rendered output. The order in the list determines which media type is more desirable than the others. The most desirable media type is located at the beginning of the list. The media type of the rendered output will be compatible to at least one media type in the list. If this method cannot create a Renderer for the specified parameters, null is returned.

The RendererFactory in Apache Clerezza is an OSGi service which manages a set of registered TypeRenderlets. The createRenderer() method of this factory creates a special Renderer based on a TypeRenderlet. The class implementing the Renderer based on TypeRenderlet is named TypeRenderletRendererImpl.

The RendererFactory uses the TypePrioritizer service to sort the “types” of the node to be rendered and find the best suitable TypeRenderlet according to the sorted types, supported mode pattern, and

supported media type pattern of registered TypeRenderlets. If more than one TypeRenderlet meet the criteria, the bundle start level of the candidate TypeRenderlets will be used to determine the most suitable one, namely the one that has the highest bundle start level.

TypeRenderlet

A TypeRenderlet generates a representation of a resource described in TripleCollections. Table 15 describes the interface definition of TypeRenderlet.

Table 15: Interface definition of TypeRenderlet.

Return Value	Method Signature and Description
`UriRef`	`getRdfType()` Returns the rdf:type supported by this TypeRenderlet
`String`	`getModePattern()` Returns the regular expression (regex) of the rendering modes supported by this TypeRenderlet.
`MediaType`	`getMediaType()` Returns the media type pattern which this TypeRenderlet supports.
`void`	`render(GraphNode node, GraphNode context, Map<String, Object> sharedRenderingValues, CallbackRenderer callbackRenderer, RequestProperties requestProperties, OutputStream os)` Renders the specified node, possibly considering the specified context of the rendering request (*e.g.*, for which user the node is to be rendered), to an OutputStream. The parameter CallbackRenderer is a renderer that can be used to render GraphNodes. Thus, in a rendering process, a TypeRenderlet may involve other renderlets to render the same or other GraphNodes. The rendered GraphNodes have the same media type as the media type of the output of the calling renderlet. The parameter sharedRenderingValues is a java.util.Map that can be used for sharing values across different renderlets involved in a rendering process. The typical use of the sharedRenderingValues is to prevent repeated computation of the same values. The parameter requestProperties contains properties of the rendering request.

As described in Table 15, a CallbackRenderer is passed to a TypeRenderlet. The CallbackRenderer is an interface specifying the following two methods to be implemented:

```
public void render(GraphNode resource, GraphNode context, String mode,
        OutputStream os) throws IOException;
public void render(UriRef resource, GraphNode context, String mode,
        OutputStream os) throws IOException;
```

The first render() method renders the specified resource and context in the specified mode to an OutputStream, whereas the latter render() method renders the specified "named resource" (non blank node) using the GraphNode obtained through the get() method of the GraphNodeProvider service (cf. Section 3.4.4).

The object of type RequestProperties passed to the render() method of a TypeRenderlet represents properties of the rendering request within which the TypeRenderlet is used. This object provides methods listed and described in Table 16.

Table 16: Methods of RequestProperties.

Return Value	Method Signature and Description
	`RequestProperties(UriInfo uriInfo, HttpHeaders requestHeaders, MultivaluedMap<String, Object> responseHeaders, String mode, MediaType mediaType, BundleContext bundleContext)` Constructs an instance of RequestProperties.
`HttpHeaders`	`getRequestHeaders()` Returns the request headers passed to the constructor.
`MultivaluedMap <String, Object>`	`getResponseHeaders()` Returns the response headers passed to the constructor.
`UriInfo`	`getUriInfo()` Returns the UriInfo passed to the constructor.
`MediaType`	`getMediaType()` Returns the media type passed to the constructor. It is the media type of the representation to be produced.
`String`	`getMode()` Returns the rendering mode passed to the constructor. It is the mode a TypeRenderlet is invoked with. This is mainly used to allow invoking a callback renderer with the inherited mode.
`<T> T`	`getRenderingService(final Class<T> type)` Returns an instance of the requested rendering service.

The getRenderingService() method of a RequestProperties is used to access a contextual rendering service by specifying its type. A rendering service is a normal OSGi service whose implementing class is annotated with @org.apache.clerezza.platform.typerendering.WebRenderingService. Such services are designed to be available to renderlets, should not have any side effects, and be oriented towards the presentation of data.

Scala Server Pages (SSP) and SspTypeRenderlet

Scala XML support allows seamless integration of XML codes with Scala. Based on this XML support in Scala, Apache Clerezza develops a templating feature called Scala Server Pages (conceptually similar to Java Server Pages), which allows for using Scala to render a response resource to a particular output format. This works by transforming a Scala Server Page file into a Scala source file and compiling it. The content of the Scala Server Page becomes the content of a method returning AnyRef. The returned Object will be transformed to a String and then to a byte-array to be written to the response stream.

The following values are available in a Scala Server Page:

- renderer: a CallbackRenderer, which can be used to delegate rendering to another TypeRenderlet. Usually not used directly, but via the render() method.
- res: the GraphNode of the response resource to be rendered, which is dynamically converted to a RichGraphNode. A RichGraphNode decorates a GraphNode with additional methods to be part of a DSL (Domain Specific Language)-style Scala library.
- context: the GraphNode with contextual information, such as description of the current user.
- mode: a String defining the rendering mode.

- uriInfo: the UriInfo of the Web request, which allows for accessing the request URI and query parameters.
- sharedRenderingValues: a java.util.Map[String, Object] used to share values across different renderlets and ScalaServerPages involved in the creation of a representation. Typically, this Map is not accessed directly, instead, values are retrieved with $("key") and are set with $("key") = newValue.

Designated OSGi services can be accessed from a Scala Server Page with the following notation: $[serviceInterface], *e.g.*, $[AdvertisingService].getBanner. Only OSGi services that are annotated with @org.apache.clerezza.platform.typerendering.WebRenderingService are accessible from a Scala Server Page.

Apache Clerezza provides the class SspTypeRenderlet, an implementation of the interface TypeRenderlet which delegates the actual rendering task to a compiled Scala Server Page. On every request the Scala Server Page is checked for changes and recompiled if needed. Therefore, the constructor of the class SspTypeRenderlet takes a URL as argument for locating the Scala Server Page file to be used:

```
SspTypeRenderlet(URL sspLocation, UriRef rdfType, String modePattern,
        MediaType mediaType, CompilerService scalaCompilerService);
```

Registering Scala Server Pages as TypeRenderlet Services

Apache Clerezza provides an OSGi service that can be located under the class name ScalaServerPagesService for registering Scala Server Pages as TypeRenderlet services. The ScalaServerPagesService implements the following method for this purpose:

```
public ServiceRegistration registerScalaServerPage(URL sspLocation,
        UriRef rdfType, String modePattern, MediaType mediaType);
```

This method will create an instance of SspTypeRenderlet using the specified SSP (sspLocation) and register this instance as a TypeRenderlet service. All registered TypeRenderlet services will be bound to the abovementioned RendererFactory.

TypeRenderletRendererImpl

The class TypeRenderletRendererImpl implements Renderer. Its constructor receives a TypeRenderlet instance and other parameters needed for implementing the render() method. The render() method delegates the rendering task to the TypeRenderlet's render() method. An instance of CallbackRenderer, RequestProperties, and Map<String, Object> for shared rendering values are created and passed to the render() method of the TypeRenderlet.

3.4.8 Authentication

Figure 13 shows the authentication process when a user tries to access a Web resource. The request is received by the Jetty Web server, and then passed to the Web Request Handler (WRHAPI framework). Jetty Web server and WRHAPI framework are third party bundles (cf. Section 2). The Web Request Handler will pass the request to the JaxRsHandler (Triaxrs) which will look for the matching JAX-RS resource method to actually process the request.

WRHAPI allows Filter services to be registered which will be invoked (if there is a Web request to be processed) before passing the request to the JaxRsHandler. The Web Request Handler does not use specific order in invoking registered Filters. A Filter must implement the following method:

```
public void handle(org.wymiwyg.wrhapi.Request request,
        org.wymiwyg.wrhapi.Response response,
        org.wymiwyg.wrhapi.Handler rest)
            throws org.wymiwyg.wrhapi.HandlerException;
```


This method handles a Web request. It may forward the processing of a request to the rest of the filter-chain by calling the handle() method of the specified Handler and passing either the original or a new request and response object as parameters. Otherwise, the rest of the filter-chain will not be applied and the response prepared by this method will be delivered to the Web server.

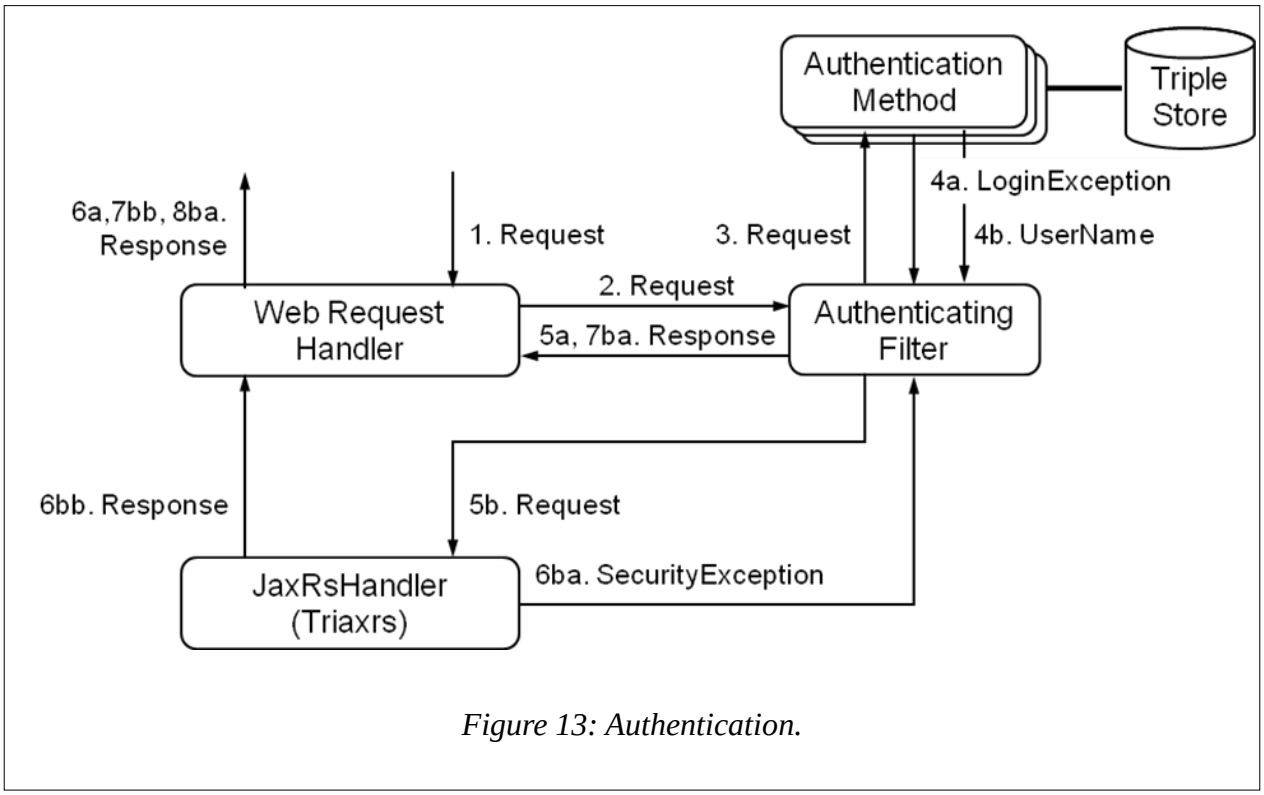

Figure 13: Authentication.

The class AuthenticatingFilter implements this Filter interface. It has the responsibility to authenticate the user who has submitted the Web request. However, it delegates the task to a set of registered AuthenticationMethods in a sequence according to their priority (weight). Each AuthenticationMethod implements the following methods:

```
public boolean authenticate(Request request, Subject subject)
        throws LoginException, HandlerException;

public boolean writeLoginResponse(Request request, Response response,
        Throwable cause) throws HandlerException;
```

The authenticate() method should return true and add a principal containing the user name and possibly some credentials to the specified javax.security.auth.Subject, if the user can be successfully authenticated. If no authentication information is available, false should be returned and the specified subject should remain unchanged. However, if authentication information is available, but the authentication failed, then a LoginException should be thrown. The writeLoginResponse() method is supposed to fill the specified Response object with information needed to prompt the user to enter login (authentication) data.

Apache Clerezza provides three AuthenticationMethod implementations: FoafSslAuthentication, CookieAuthentication (cookie-based authentication), and BasicAuthentication (HTTP basic authentication). In case of HTTP basic authentication, if the Web request does not contain the user credentials, the UNAUTHORIZED response status code is sent to the client. In case of cookie-based authentication, a failed authentication leads to a redirection to a login page.

The AuthenticatingFilter invokes the authenticate() method. If a LoginException is catched, it will invoke the writeLoginResponse() method of the currently used AuthenticationMethod. The resulting response will be passed to the Web Request Handler. If this AuthenticationMethod returns false in its writeLoginResponse() method, the AuthenticatingFilter will try to invoke the

writeLoginResponse() method of each AuthenticationMethod starting from the one with the highest priority until it gets a true in the return value or no more AuthenticationMethod is available.

If the authenticate() method returns false, the handle() method of the specified Handler will be invoked in the context of an anonymous user. Otherwise, the handle() method will be invoked in the context of the authenticated user. To invoke a method in the context of a particular Subject, the following construct is used:

```
Subject.doAsPrivileged(subject, new PrivilegedExceptionAction() {
    @Override
    public Object run() throws Exception {
        // method invocation
        return null;
    }
}, null);
```

The doAsPrivileged() method requires three parameters: a subject, an action, and an access control context. The specified action is carried out as the specified subject within the specified access control context. This means, Triaxrs processes the Web request on behalf of the subject (authenticated or anonymous user). Doing this is necessary to enable checking the rights of a subject to perform a particular action, as described in the next section.

As opposed to Figure 13, the JaxRsHandler is not directly invoked by the AuthenticatingFilter, but actually through a filter-chain provided by the Web Request Handler to the AuthenticatingFilter. Drawing these details into the figure would just overload it and make it difficult to understand. Now if the resource requested by the user needs an access control, and the JAX-RS resource method found by the JaxRsHandler to handle this request throws a SecurityException due to the missing user rights, then the AuthenticatingFilter will invoke the writeLoginResponse() method of each AuthenticationMethod starting from the one with the highest priority until it gets a true in the return value or no more AuthenticationMethod is available.

CookieLogin

The class CookieLogin provides a Web service for registered users to login to the platform. It is a JAX-RS root resource class annotated with @Path("login"). It implements a GET method to deliver a login page and a POST method for the clients to send user name and password.

```
@GET
public GraphNode loginPage(@Context UriInfo uriInfo,
        @QueryParam("referer") String refererUri,
        @QueryParam("cause") Integer cause)

@POST
public Object login(@FormParam("user") final String userName,
        @FormParam("pass") final String password,
        @FormParam("referer") final String referer,
        @DefaultValue("false")
        @FormParam("stayloggedin") final Boolean stayLoggedIn,
        @Context final UriInfo uriInfo)
```

The writeLoginResponse() method of the class CookieAuthentication creates a response that redirects a browser to the path /login. The status code used is 307 (Temporary Redirect) and the referer query parameter points to the original request URL.

3.4.9 Authorization

Apache Clerezza implements authorization based on Java Authentication and Authorization Service (JAAS). Upon startup of the platform, the default Security Manager provided by the Java Virtual Machine is activated. The class org.apache.clerezza.platform.security.SecurityActivator is an OSGi component which upon its activation installs a user-aware java.security.Policy. Apache Clerezza provides a class called UserAwarePolicy which extends java.security.Policy and overrides its

getPermission() method.

```
@Override
public PermissionCollection getPermissions(final ProtectionDomain domain) {
    PermissionCollection result;
    Principal[] principals = domain.getPrincipals();
    if (principals.length > 0) {
        final Principal user = principals[0];
        result = getUserPermissionsFromSystemGraph(user);
    } else {
        result = originalPolicy.getPermissions(domain);
    }
    return result;
}
```

The getPermissions() method will be invoked by the Java security framework when checkPermission() is called to enforce access control. The ProtectionDomain passed to this method contains information on the authenticated user. This method retrieves permissions assigned to this user from the System Graph and returns them. The originalPolicy in the above code snippet is assigned the result of Policy.getPolicy() in the UserAwarePolicy's constructor method.

As depicted in Figure 14, before a service executing a code segment that requires an authorization, *e.g.*, when a user wants to modify an MGraph, an access control is triggered by invoking the method AccessController.checkPermission() and passing the respective Java Permission object as the parameter (in this case, a Java Permission object for modifying an MGraph). Since the code segment is performed on behalf of the authenticated subject, JAAS consults (retrieve permissions of a subject from) currently installed security policy, in order to check, whether the subject has the requested permission.

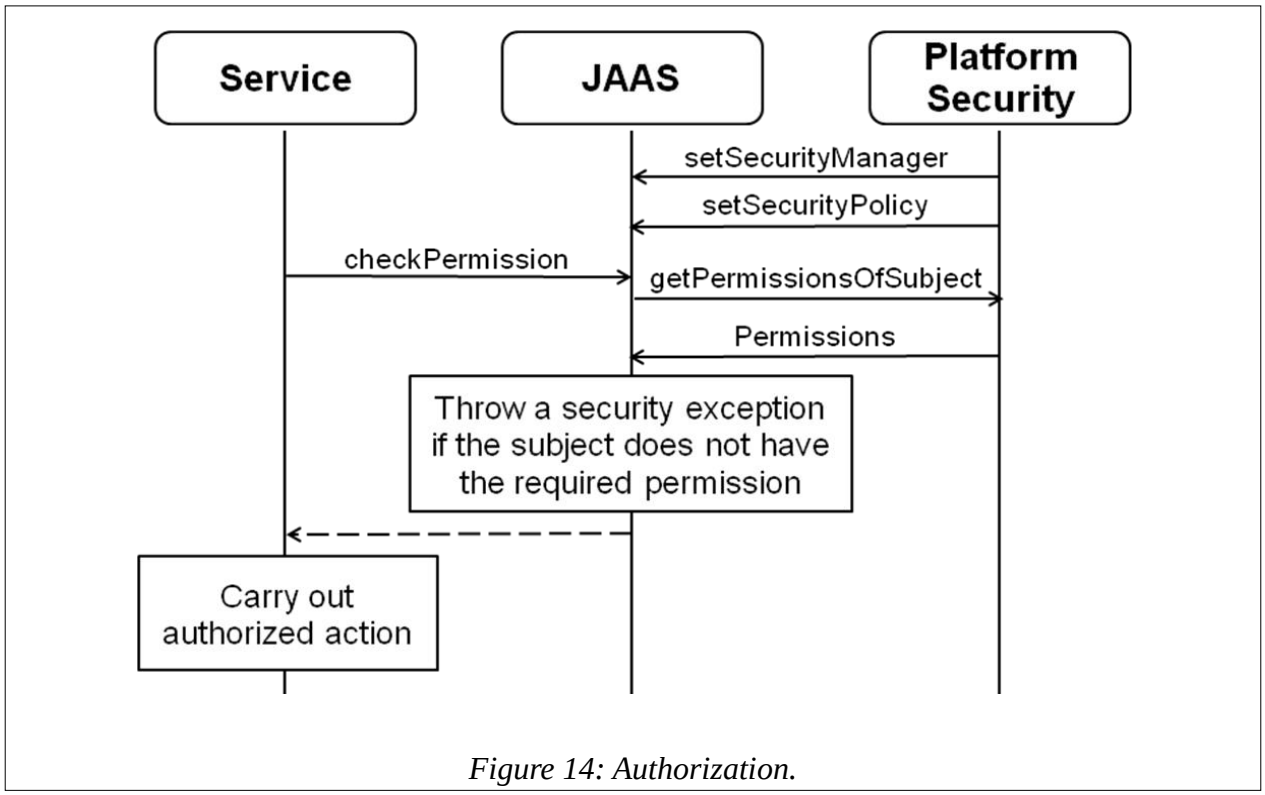

Figure 14: Authorization.

4 SciMantic Specific Bundles

SciMantic specific bundles implement a knowledge management and sharing functionality for a Knowledge Sharing System (KSS). A KSS instance offers this functionality to a set of registered users. KSS instances are supposed to be connected to one or more Knowledge Sharing Networks (KSNs). A KSN is dedicated to sharing knowledge units of specific areas of interest determined by its community. A KSS instance can join several KSNs determined by the operator of the KSS instance. Figure 13 depicts a high level architecture view of a KSS.

In SciMantic, a knowledge unit is the smallest globally identifiable pieces of knowledge. A knowledge unit can be a complex conglomerate of resources and references, but to a knowledge management system it is an inseparable unit. To illustrate this on an example, let us assume a knowledge unit to be a scientific report. But it can also be any other type of information such as an audio file, video file, or a Web page. A report contains among other things: structured text, formatting information, figures, a list of keywords, references, and meta information. The knowledge management system treats all this information as a unit and assigns a globally unique

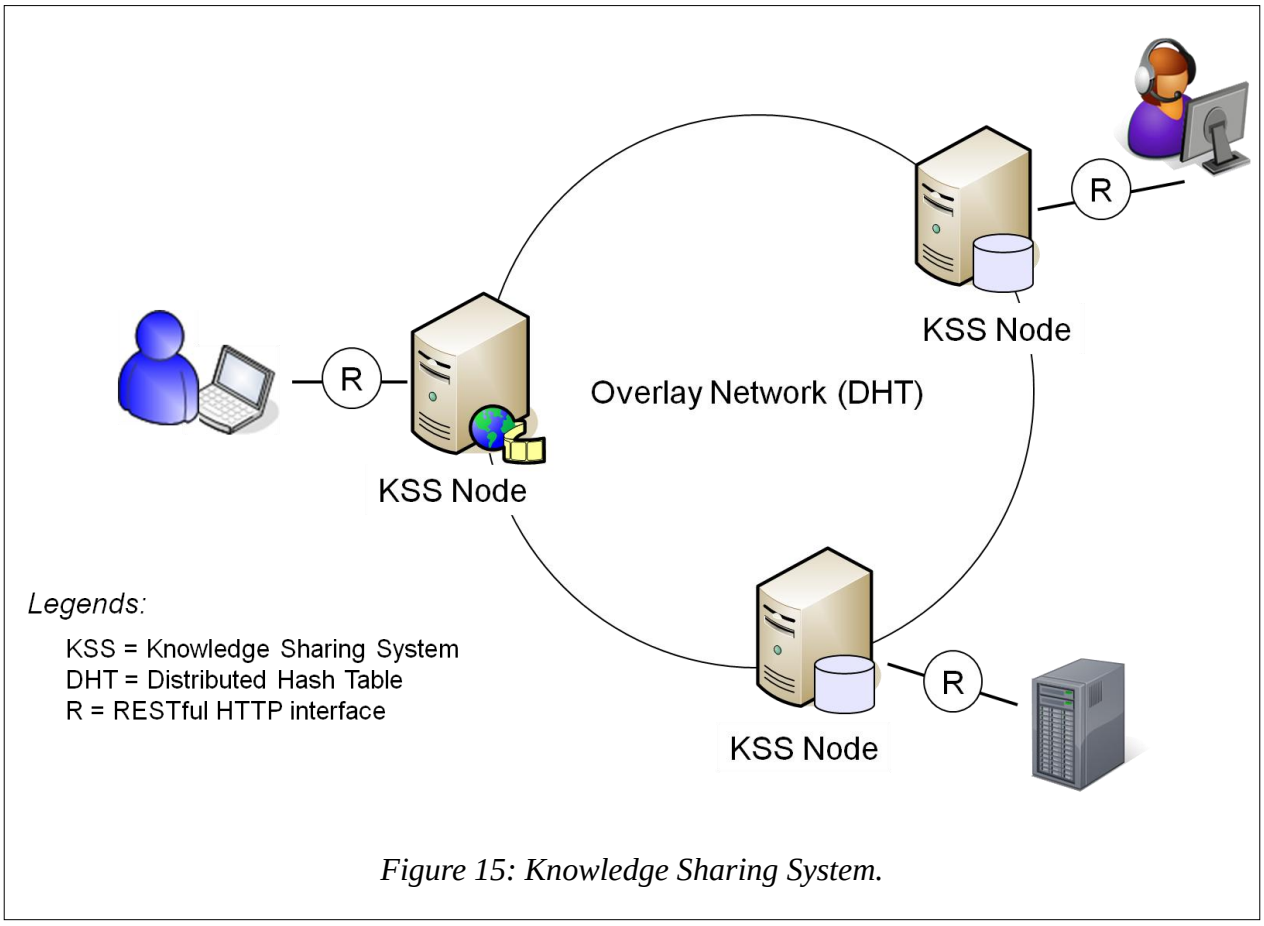

Figure 15: Knowledge Sharing System.

identifier to each knowledge unit that is entered into its system. Furthermore, in order to enhance searching possibilities, knowledge units are tagged with keywords.

4.1 Knowledge Unit Identifier

A candidate for knowledge unit identifier is URL. This has the advantage that it can be used directly to *access* a knowledge unit in a Web browser's URL input field. The problem lies in the management of *shared* knowledge units. Replicating knowledge units becomes difficult, because they should have the same identifier (URL), but must be located on different machines. Thus, to still be able to use URL and at the same time support replication, a convention to use only a part of a

URL to identify a knowledge unit is necessary. In SciMantic, a knowledge unit is identified only by the path part of a URL. The scheme and authority part can be different for the same knowledge unit. Furthermore, to distinguish a knowledge unit from its replications, a special suffix can be appended to the original URL. For example, adding the suffix "-replica-1" to a knowledge unit URL means that it is a replication number 1.

4.2 Ontology

The following classes and properties are used to describe a knowledge unit:

- rdf:type
- scimantic:KnowledgeUnit: the rdf:type of a resource denoting that it is a knowledge unit
- dcterms:creator: points to the owner of the knowledge unit
- dcterms:created: points to the creation time of the knowledge unit
- dcterms:modified: points to the modification time of the knowledge unit
- scimantic:shared: points to the knowledge sharing network the knowledge unit is shared
- scimantic:tag: points to a tag
- scimantic:Tag: the rdf:type of a tag
- scimantic:weight: points to the weight of a tag
- dc:subject: points to the skos:Concept of a tag

Note that the creator of a knowledge unit might not be the same as its author. The authors of a knowledge unit as well as other meta information may be added in future for the support of faceted search.

In order to bridge language differences and to enable extended reasoning, keywords are modeled as resources representing a concept and they receive a URI in the scimantic namespace. Then labels (the actual keywords) can be added in different languages and alternative spellings. This also avoids confusion between keywords that are spelled the same but refer to different concepts (such as apple fruits and Apple computers). Additionally, concepts can be interlinked and describe their relations to each other. This may be used in the future to provide not only keyword search, but also to find related knowledge units using the context of keyword concepts. This gives an impression of how this application can benefit from Semantic Web technologies. The W3C defined the Simple Knowledge Organization System (SKOS) vocabulary to describe such concepts and their relations in RDF.

4.3 Knowledge Unit Management Service

SciMantic provides a Knowledge Unit Management service for creation, read, update, and deletion (CRUD) of knowledge units as listed in Table 17. The root resource is annotated with @Path("/kms/ku"). Only registered users can create knowledge units. Being an owner of a knowledge unit implies the possession of any rights on that knowledge unit. Users who are not the owner of a knowledge unit cannot share that knowledge unit. Only a knowledge unit marked as shared can be read by other users. Finally, a user who possesses java.security.AllPermission has the rights to perform any action on any knowledge units. Note that, the scope of the rights a user has are within the platform instance the user is registered to.

Table 17: Web service for managing knowledge units.

Return Value	Method Signature and Description

TripleCollection	`@GET`
	`@Produces({"*/*"})`
	`@Path("{id:.*}")`
	`getKnowledgeUnit(@PathParam("id") String id)`
	Returns a TripleCollection for the knowledge unit identified by the request URI.
	This method throws a WebApplicationException containing
	• NOT FOUND (404) if the knowledge unit cannot be found.
	• UNAUTHORIZED (401) if authorization is required.

Response	`@PUT`
	`@Path("{id:.*}")`
	`createKnowledgeUnit(@PathParam("id") String id,`
	`TripleCollection knowledgeUnit)`
	Creates a knowledge unit for the specified URI using the specified TripleCollection if the user has the required rights. This method can be used to replace a locally existing knowledge unit with the same identifier if the user has the required rights or if she is the owner of the knowledge unit to be replaced.
	This method returns a response with status code:
	• CREATED (201) if a knowledge unit is newly created.
	• OK (200) if the knowledge unit can be updated.
	• BAD REQUEST (400) if the required data are missing.
	• UNAUTHORIZED (401) if authorization is required.

Response	`@DELETE`
	`@Path("{id:.*}")`
	`deleteKnowledgeUnit(@PathParam("id") String id)`
	Deletes a knowledge unit with the specified identifier if the user has the required rights.
	This method returns a response with status code:
	• NO CONTENT (204) if the knowledge unit can be deleted.
	• NOT FOUND (404) if the knowledge unit cannot be found.
	• UNAUTHORIZED (401) if authorization is required.

4.4 Knowledge Unit Tagging Service

The class KnowledgeUnitTagging implements the tagging service. It is a JAX-RS root resource annotated with @Path("/ku-tag"). This Web service provides the following HTTP POST method to add tags to or remove tags from a knowledge unit:

```
@POST
@Consumes("multipart/form")
@Path("update-tags")
public Response updateTags(MultiPartBody form);
```

The parameter form contains the UriRef of the knowledge unit, an optional set of skos:Concept and the corresponding weight to be used to tag the knowledge unit, and another optional set of skos:Concept which determines the tags to be removed from the knowledge unit.

4.5 Knowledge Unit Sharing Service

The class KnowledgeUnitSharing implements the sharing service. It is a JAX-RS root resource

annotated with @Path("/ku-share"). This Web service provides the following HTTP POST method to share and unshare a knowledge unit:

```
@POST
public Response shareKnowledgeUnit(@QueryParam("id") String id,
        @QueryParam("network") String network,
        @QueryParam("share") Boolean share);
```

The parameter network specifies the knowledge sharing network where the specified knowledge unit is to be shared or unshared. SciMantic uses P2P technology to connect KSS instances. Therefore, KSS instances are modeled as peers (nodes) participating in a P2P overlay network. Current implementation of this overlay network is based on Distributed Hash Tables (DHT). The only infrastructure a KSN must provide is a bootstrapping mechanism that allows new peers to join an existing network. All the other mechanisms are provided for by the participating peers.

An entry of a shared knowledge unit in a DHT has the following key-value-pair:

- Key = Hash(KU-Id)
- Value = URL of the knowledge unit

In case of replication support, the DHT must store several values for the same key. In order to support keyword searching, the DHT is also used to store the following key-value-pair for a knowledge unit which is tagged with a keyword identified by a UriRef C of a skos:Concept:

- Key = Hash(C)
- Value = URL of the knowledge unit

Thus, the URL of all knowledge units tagged with C will be stored at a certain peer.

4.6 Knowledge Unit Searching Service

As described in Section 4.5, to make use of the keyword tags, a dedicated searching service builds a distributed keyword index and allows users to search for knowledge units with keywords. Without a keyword-based search mechanism, a user has to know the identifier of a knowledge unit in order to access it. An identifier is long and possibly cryptic, and therefore does not represent an appropriate mean for user interactions.

To search for knowledge units with a keyword, the URI of the keyword concept has to be hashed. The resulting key can be used to fetch the list of knowledge units tagged with the keyword concept. More complex queries fetch multiple candidate-lists and process them locally to produce a final list. A search for knowledge units tagged with keyword x and y retrieves a candidate-list with hash(x) and another one with hash(y). In the final result only those knowledge units appear, whose URIs are contained in both candidate-lists.

SciMantic provides a JAX-RS root resource class, KnowledgeUnitSearching, annotated with @Path("/ku-search") which implements the following method:

```
@POST
public TripleCollection searchKnowledgeUnit(@QueryParam("query") String query);
```

4.7 Knowledge Unit Update Event Service

In order to allow users to get notified if a knowledge unit is modified or if knowledge units tagged with certain keywords are updated, SciMantic provides a Knowledge Unit Update Event subscription and notification service. In this respect, two roles are distinguished: publisher and subscriber. A publisher is a KSS instance whose registered users allow other users to be notified when a knowledge unit is created or updated. A subscriber is a KSS instance whose registered users want to get notified of knowledge unit updates. Table 18 and Table 19 describe those methods

supported by a publisher respectively a subscriber. The JAX-RS root resource class KnowledgeUnitUpdateEvent, which is annotated with @Path("/ku-update-event"), implements all these methods.

Table 18: Methods supported by a publisher.

Return Value	Method Signature and Description

Response	`@POST`
	`@Path("inter-kss/subscribe")`
	`interKssSubscribe(@FormParam("url") UriRef subscriberUrl,`
	`@FormParam("userName") String userName,`
	`@FormParam("kUnits") List<UriRef> kUnits,`
	`@FormParam("concepts") List<UriRef> concepts)`
	This method registers the specified subscriber's URL and user name for the specified list of knowledge units and concepts (keywords). If there is an update to the affected knowledge units (specified ones or those tagged with the specified keywords), a POST request will be sent to the subscriber's URL containing the specified user name and a list of affected knowledge units as request parameters. The subscription information is stored in a subscription graph.

Response	`@POST`
	`@Path("inter-kss/unsubscribe")`
	`interKssUnsubscribe(@FormParam("url") UriRef subscriberUrl,`
	`@FormParam("userName") String userName,`
	`@FormParam("kUnits") List<UriRef> kUnits,`
	`@FormParam("concepts") List<UriRef> concepts)`
	This method unregisters the specified subscriber's URL and user name for the specified list of knowledge units and concepts (keywords).

Table 19: Methods supported by a subscriber.

Return Value	Method Signature and Description

Response	`@POST`
	`@Path("subscribe")`
	`subscribe(@FormParam("kUnits") List<UriRef> kUnits,`
	`@FormParam("concepts") List<UriRef> concepts)`
	This method sends on behalf of the requesting user a subscription request to the publisher. In the subscription request, the subscriber's URL, i.e. http://<authority>/ku-update-event/inter-kss/notify is specified. If there is an update to the affected knowledge units (specified ones or those tagged with the specified keywords), a notification will be sent to the subscriber's URL.

Response	`@POST`
	`@Path("unsubscribe")`
	`unsubscribe(@FormParam("kUnits") List<UriRef> kUnits,`
	`@FormParam("concepts") List<UriRef> concepts)`
	This method sends on behalf of the requesting user a request to the publisher to unsubscribe the update event for the specified knowledge units and concepts (keywords).

Response	`@POST`
	`@Path("inter-kss/notify")`
	`interKssNotify(@FormParam("userName") String userName,`
	`@FormParam("kUnits") List<UriRef> kUnits)`
	This method receives notification about knowledge unit update events. The parameter username specifies the user who wants to get this notification from the KSS instance. The parameter kUnits contains a list of knowledge units which

	have been updated or newly created. This notification information is stored in the user's content graph, which can be fetched later by the user herself (on demand).

The class KnowledgeUnitUpdateEvent also provides the OSGi service KnowledgeUnitUpdateNotifier which implements the following method:

```
public void notifySubscribers(Set<GraphNode> updatedKnowledgeUnits);
```

The method notifySubscribers() may be invoked by the Knowledge Unit Management, Knowledge Unit Tagging, or Knowledge Unit Sharing service.

4.8 Knowledge Unit Publisher Discovery Service

A DHT is used to store key-value pairs comprising publishers' data:

- Key = Hash(C), where C is the UriRef of a skos:Concept representing the respective keyword that can be subscribed.
- Value = URL of the subscription service provided by the publisher, i.e. http://<authority>/ku-update-event/subscribe.

The class KnowledgeUnitPublisherDiscovery provides an OSGi service implementing the methods listed in Table 20. It is also a JAX-RS root resource class annotated with @Path("/ku-publisher-discovery").

Table 20: Methods supported by the Knowledge Unit Publisher Discovery service.

Return Value	Method Signature and Description

`Response`	`@POST`
	`@Path("register")`
	`registerPublisher(@FormParam("concepts") List<UriRef> concepts)`
	This method stores for each concept in the specified list, a key-value pair in a DHT, where the key is the hash value of the concept URI and the value is the URL of the subscription service provided by the KSS instance.

`Response`	`@POST`
	`@Path("deregister")`
	`deregisterPublisher(@FormParam("concepts") List<UriRef> concepts)`
	This method removes the stored registration from the DHT for each of the specified concepts.

`TripleCollection`	`@GET`
	`retrievePublishers(@FormParam("concepts") List<UriRef> concepts)`
	Retrieves all subscription services for the specified list of concepts. Each triple in the returned TripleCollection relates a concept as subject to a subscription service as object with the predicate scimantic:subscriptionService.

4.9 DHT Service

SciMantic defines a DhtProvider service interface (cf. Table 21) to allow implementations to offer distributed indexing functionality, i.e. to enable knowledge units to be shared, indexed, and searched using keywords. The DhtProvider interface hides the DHT service implementation from the rest of the system, thus, allowing the service to be easily changed. The DhtProvider interface defines a very basic DHT interface that can be fulfilled by any DHT service. A KSS should not depend on a concrete DHT service, because it may become necessary to exchange the DHT service

in the future.

OpenChordProvider is an OSGi service that implements the DhtProvider interface and offers a DHT API using the OpenChord DHT implementation. Multiple DhtProviders can coexist, however, currently there is no mechanism to prefer one implementation over the other at runtime; but such functionality can be implemented. The OpenChordProvider service depends on the third party bundle openchord that exports the OpenChord implementation.

Table 21: DhtProvider interface definition.

Return Value	Method Signature and Description
`void`	`setup(URL configuration)` Triggers necessary setup operations using the specified configuration.
`void`	`shutdown()` Performs necessary clean-up operations to shut the DhtProvider down.
`void`	`join(URL bootStrap)` Joins an overlay network through the bootstrap URL.
`void`	`leave()` Leaves the overlay network the DhtProvider is currently connected to.
`void`	`put(String key, Serializable value)` Inserts a key-value pair into the DHT.
`Serializable[]`	`get(String key)` Retrieves values which are saved into the DHT with the specified key.
`void`	`remove(String key, Serializable value)` Removes the specified key-value pair from the DHT.

5 Evaluation

This section evaluates the implemented KSS. It aims to identify performance bottlenecks, discuss implementation aspects that affect performance in the application and to demonstrate how the application scales. The KSS services are not computationally intensive and are not expected to become bottlenecks of the implementation. Performance measurements show that the overhead added by the knowledge unit searching service when retrieving knowledge units from the DHT can be neglected. It is indistinguishable from the variability of the data set.

The areas that deserve most attention in this evaluation are the Web service implementation, the DHT service, and the DHT organization. The Web service is the interface to the outside and difficult to control. If a Web service is popular, it will require considerable resources on the server to serve all clients. There are not many possibilities to avoid this completely, but the service can be designed with efficiency in mind. The DHT on the other hand is the most complex component of the KSS solution. When data is entered, it gets transmitted over a network to other computers. The task of routing to the desired computers in the overlay network and then to establish a connection for data transfer produces delay times that differ in magnitudes from local processing times. The manner in which the data is organized in the DHT directly affects the number of interactions the DHT service needs to initiate with other nodes. The expectation is that, this is the most likely bottleneck.

5.1 Performance Analysis of Web Interfaces

Because the Web services are implemented according to the REST architectural style, we can expect them to fare well in terms of performance. RESTful Web services are stateless and tend to scale well as a result. Stateless services relieve the server from keeping state information for every connection with a client. Keeping state information for a single connection is not necessarily complex but the effort scales proportionally to the number of connections. Most clients have sufficient processing power today. Thus, to prevent the server from becoming a bottleneck, it is beneficial to let the client deal with state information where possible. The server can concentrate on providing the Web services and has as many resources as possibly available for that task. The average time needed by the implemented Web services to process a single request is around 10 ms on an Intel Core 2 Duo CPU at 2.8 GHz with 4 GB RAM. If 1000 requests need to be processed in parallel, this average time increases to 400 ms. In order to avoid network delay factor, the server under test was connected directly to the client.

5.2 Performance Analysis of the KSS DHT Service

Chord lookup algorithm has been shown to be efficient and scalable [13]. It requires **O**(log N) messages per lookup and **O**(log N) state per node, where N is the number of nodes. However, to insert a knowledge unit tagged with k keywords, k+1 DHT lookups have to be performed. With respect to knowledge unit retrieval, the number of lookups depends on the number of keywords searched. Afterwards, the results of each lookup need to be combined to meet the filter conditions of the query. This additional processing time is negligible compared to the total lookup time. But there is a performance bottleneck in the DHT service because it works synchronously. It only opens a single connection at a time and waits for it to be completed before it opens a new connection. OpenChord supports asynchronous interaction with the DHT, but the KSS implementation has not implemented this so far. Therefore, the time required to insert or retrieve knowledge units increases linearly with the increase of the number of keywords used.

5.3 Scalability Evaluation

In order to assess the scalability of the KSS, the evaluation focuses on measuring the change of the duration of specific tasks. Either the number of knowledge units or the number of nodes in the DHT grows during the scenario. The absolute duration value is not relevant to the evaluation. The evaluation is interested in its growth only. The tasks are insertion of data into the DHT and fetching of data from the DHT.

The evaluations were run on a single physical node. OpenChord offers a protocol that allows a DHT to communicate inside a single Java virtual machine, bypassing the network interface. This has been used because it allows great flexibility in number of nodes and the results are not affected by network transmission time. The DHT mechanisms are identical to using the TCP sockets protocol otherwise.

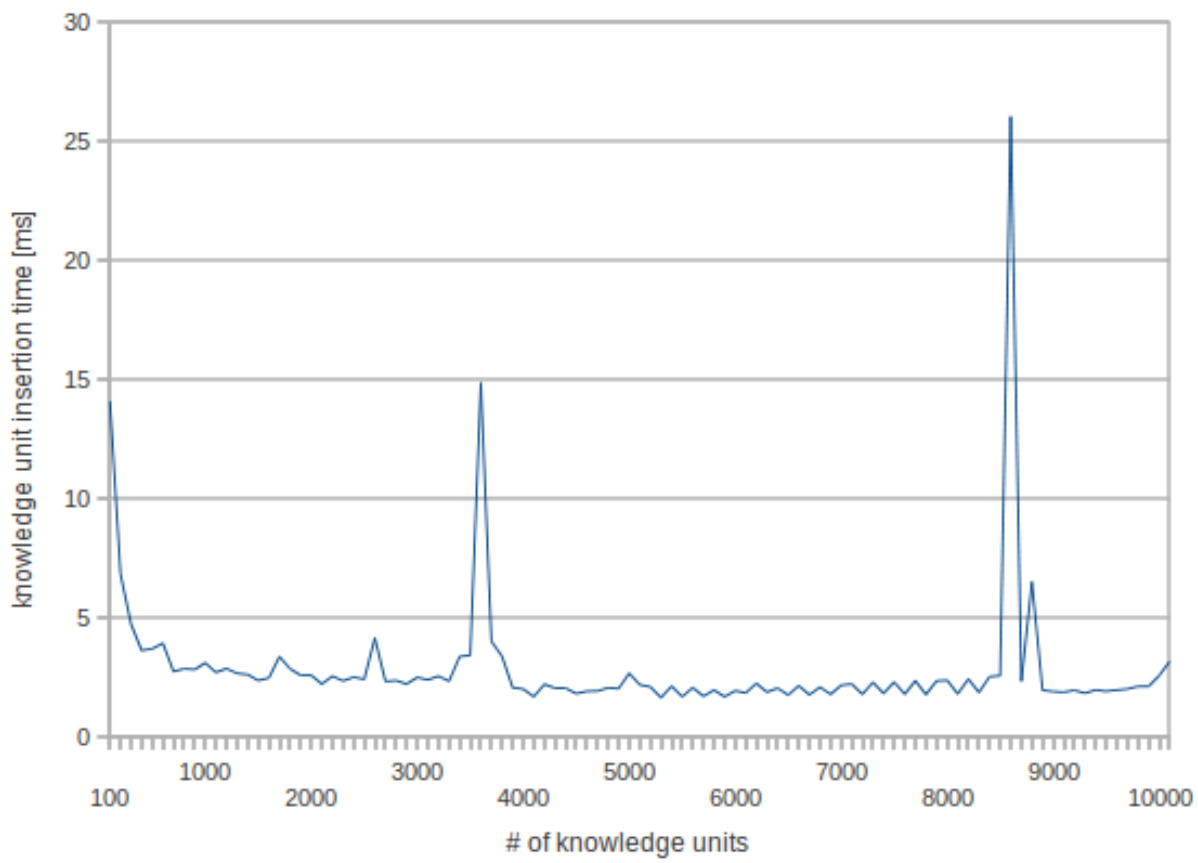

Figure 16: Average time to insert a knowledge unit as a function of the number of knowledge units already present in DHT.

Figure 14 shows the average time needed to insert a new knowledge unit into the DHT in terms of knowledge units already present in the DHT, whereas Figure 15 shows the average time to retrieve a knowledge unit from the DHT. In the tests, the routing hops remain constant. The DHT has 10 nodes which suggests that on average one routing hop will be sufficient. The evaluations have been run with 100 and 1000 nodes as well but the graphs appeared very similar. The graph shows how the DHT behaves when more and more knowledge units are saved in it. The distribution of the knowledge units over the DHT nodes is uniform. For the range of knowledge units used in the tests, the DHT is not becoming slower from being filled up (note that a single node has to save up to 1000 knowledge units). The reason for the trend of becoming faster as depicted on the graphs is not clear. It might be an unrelated side effect (*e.g.*, optimization of memory access by the operating system).

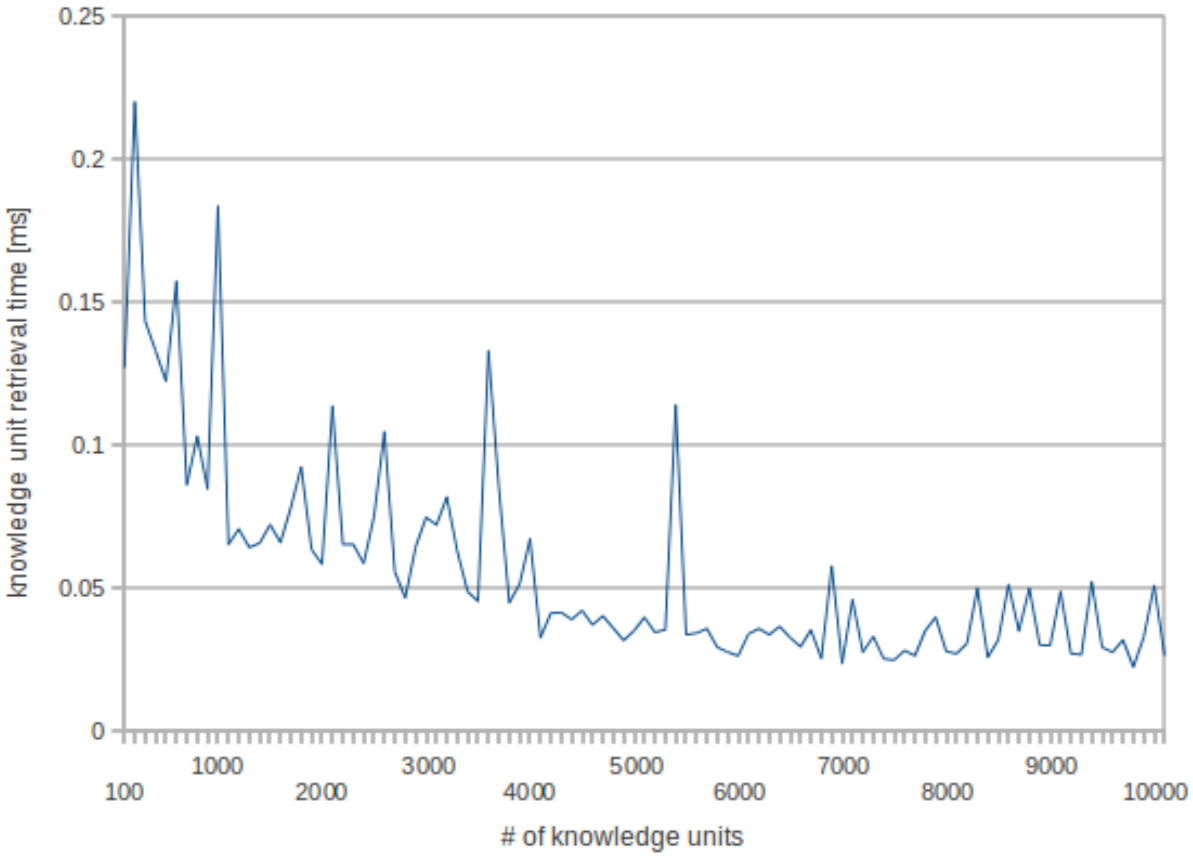

Figure 17: Average time to retrieve a knowledge unit as a function of the number of knowledge units already present in DHT.

Theoretically, all Chord DHT operations should scale logarithmically to the number of nodes [13]. Figure 16 and Figure 17 show that the average time to insert and retrieve a knowledge unit is growing with a growing number of nodes. The trend seems to suggest that the growth may be logarithmic. Evaluations with more nodes and data points would be helpful to decide whether the growth is linear or logarithmic, but the available test environment can not deal with larger numbers. A scale of 10 to 1000 nodes seems to be a realistic measure for KSN membership in a useful real life scenario.

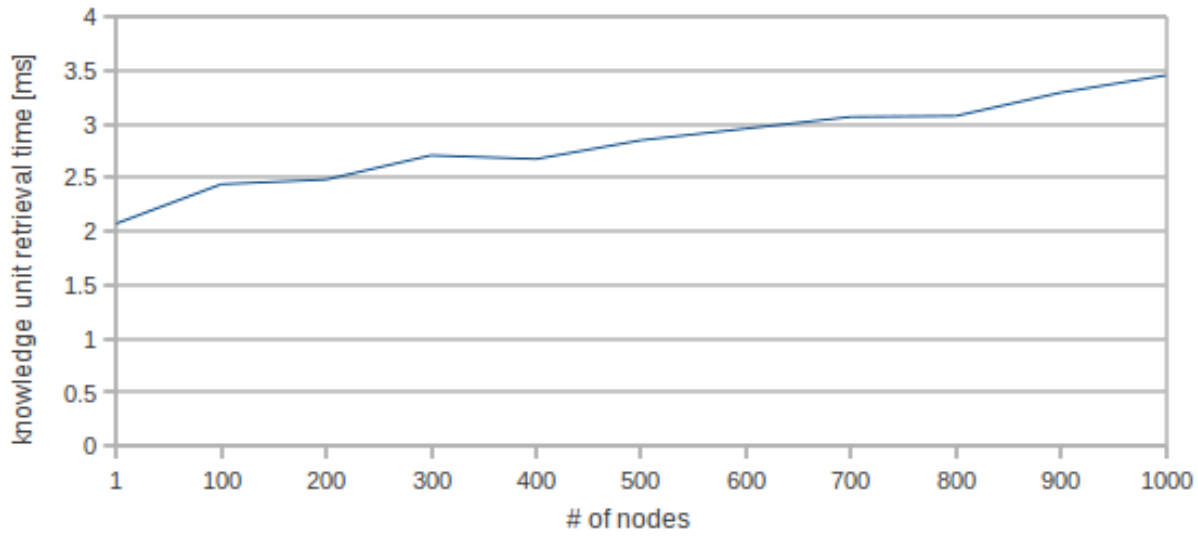

Figure 18: Average time to insert a knowledge unit as a function of the number of nodes in DHT.

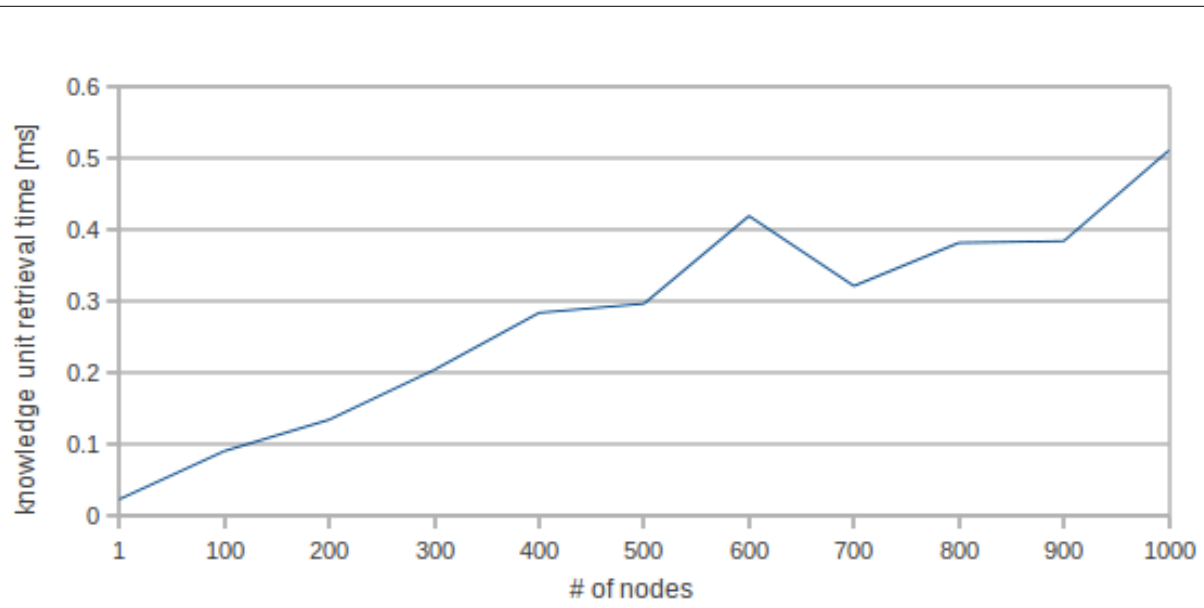

Figure 19: Average time to retrieve a knowledge unit as a function of the number of nodes in DHT.

6 Summary

This deliverable presents implementation detail of a Semantic Content Sharing System (SCSS) developed within this project. The implementation is based on the design described in previous deliverable. The SCSS is a Java program which integrates an OSGi Framework and a set of OSGi bundles to provide the required functionality. These OSGi bundles can be divided into 4 groups: third party bundles, Apache Clerezza bundles, SciMantic specific bundles, and application bundles. Since the main goal of SciMantic is the provision of a software platform for developing Web-based applications which combines Semantic Web and Peer-to-Peer (P2P) technologies, this deliverable focuses on the development of Apache Clerezza bundles and SciMantic specific bundles. Apache Clerezza has been initiated by Trialox AG to promote work within this project and to obtain support from the open source community.

Apache Clerezza bundles are classified into five groups based on their functionality in order to ease organization: Smart Content Binding (SCB), Triaxrs, Utilities, Static Web, and Platform. SCB bundles provide functionality for accessing and manipulating RDF graphs. It supports graph locking to prevent concurrent modifications and security to prevent unauthorized access. Triaxrs bundles implement JSR-311 (JAX-RS) 1.0 specification. While Utilities bundles provide for commonly needed functions, Static Web bundles provide access to a collection of static Web resources, such as commonly needed JavaScripts and stylesheets. Finally, Platform bundles constitute the largest set of bundles with diverse functions to be provided, including the Type Handling and Type Rendering mechanism introduced in Deliverable D2.1.

SciMantic specific bundles comprises bundles providing a DHT service, services for sharing content, keyword-based distributed indexing and searching, as well as services for content update event subscription and notification. These services apply P2P lookup based on OpenChord implementation. It is the implementation of these services whose performance and scalability was evaluated and reported in this deliverable. The average time needed by the implemented Web services to process a single request is around 10 ms on an Intel Core 2 Duo CPU at 2.8 GHz with 4 GB RAM. If 1000 requests need to be processed in parallel, this average time increases to 400 ms. With respect to the scalability evaluation of the implemented DHT service, the average time to insert or retrieve a content does not grow with increasing number of knowledge units present in the DHT, but grows linearly with the increase of the number of nodes in the DHT.

Concluding, the combination of P2P and Semantic Web technology is able to support implementations of interesting scenarios, *e.g.*, knowledge sharing, in a scalable manner with good performance, assuming a reasonable size of P2P nodes participating in the community and that the required services for such scenarios are generally not computationally intensive.

References

[1] Ben Adida, Mark Birbeck: *RDFa Primer, Bridging the Human and Data Webs*; W3C Working Group Note, October 2008.

[2] Aduna B.V.: *User Guide for Sesame 2.3*; 2010, URL: http://www.openrdf.org/doc/sesame2/users/.

[3] Jeremy J. Carroll: *Signing RDF Graphs*; Lecture Notes in Computer Science, Vol. 2870, Springer Verlag, September 2003.

[4] Marc Hadley, Paul Sandoz: *JAX-RS: Java API for RESTful Web Services*; Version 1.1, September 2009.

[5] Hasan (Editor): Functional Requirements and Analysis of Mechanisms for a Semantic Content Infrastructure; Deliverable D2.1, Project SciMantic, March 2010.

[6] Hasan (Editor): Architecture of a Secure and Scalable Semantic Content Infrastructure; Deliverable D2.2, Project SciMantic, September 2010.

[7] Jena Development Team: *Jena – A Semantic Web Framework for Java*; 2010, URL: http://jena.sourceforge.net/documentation.html

[8] Graham Klyne (Editor), Jeremy J. Carroll (Editor): *Resource Description Framework (RDF): Concepts and Abstract Syntax*; W3C Recommendation, February 2004.

[9] OASIS: *SCA Service Component Architecture: Assembly Model Specification*; SCA Version 1.00, March 2007.

[10] Oracle Corporation: *Java™ Authentication and Authorization Service (JAAS) Reference Guide*; 2006, URL: http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html

[11] The OSGi Alliance: *OSGi Service Platform Core Specification*; Release 4, Version 4.2, June 2009.

[12] The OSGi Alliance: *OSGi Service Platform Service Compendium*; Release 4, Version 4.2, August 2009.

[13] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan: *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*; ACM SIGCOMM 2001, San Diego, CA, August 2001.

[14] Giovanni Tummarello, Christian Morbidoni, Paolo Puliti, Francesco Piazza: *Signing Individual Fragments of an RDF Graph*; 14th International World Wide Web Conference WWW2005, Poster track, Chiba, Japan, May 2005.

[15] W3C: *SKOS Simple Knowledge Organization System Primer*; W3C Working Group Note, August 2009.

[16] W3C: *SPARQL Query Language for RDF*; W3C Recommendation, 2008.

[17] http://felix.apache.org/site/index.html

[18] http://lucene.apache.org/java/docs/index.html

[19] http://lucene.apache.org/java/3_0_1/queryparsersyntax.html

[20] http://www.scala-lang.org/

Acknowledgments

The authors would like to thank all participants of the SciMantic project who have contributed to this deliverable through constructive discussions and feedback. Many thanks to Agron Limani who has provided valuable performance test results. This project is funded by The Innovation Promotion Agency CTI of the Swiss Federal Office for Professional Education and Technology (Das Bundesamt für Berufsbildung und Technologie BBT).

List of Figures

Figure 1: A high level overview of the implementation architecture of the SCSS. 1
Figure 2: Apache Clerezza bundles. 4
Figure 3: Smart Content Binding bundles 5
Figure 4: SCB node types. 6
Figure 5: SCB graph abstractions. 7
Figure 6: Lockable MGraph. 9
Figure 7: Secured TripleCollection. 10
Figure 8: TripleCollection Providers 11
Figure 9: Modeling various types of SPARQL queries. 14
Figure 10: OSGi Services required by the TcManager. 16
Figure 11: Web Bundles. 28
Figure 12: Subclasses of DiscoBit. 30
Figure 13: Authentication 39
Figure 14: Authorization. 41
Figure 13: Knowledge Sharing System. 42
Figure 14: Average time to insert a knowledge unit as a function of the number of knowledge units already present in DHT. 50
Figure 15: Average time to retrieve a knowledge unit as a function of the number of knowledge units already present in DHT. 51
Figure 16: Average time to insert a knowledge unit as a function of the number of nodes in DHT. . 52
Figure 17: Average time to retrieve a knowledge unit as a function of the number of nodes in DHT. 52

