Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer zfi-msc-2009-09

Master Thesis

im Studiengang "Angewandte Informatik"

Query-Brokering in Semantic-Web-Umgebungen

Heiko Vollmann

Arbeitsgruppe für

Datenbanken und Informationssysteme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

September 29, 2009

Georg-August-Universität Göttingen
Zentrum für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel.	+49 (5 51) 39-17 20 10
Fax	+49 (5 51) 39-14 69 3
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den September 29, 2009

Abstract

The Semantic Web will consist of a large number of nodes. In contrast to the present web, these nodes will be capable of communicating with each other. The nodes will, for example, be able to send messages to other nodes informing them of updates or changes. They will also be able to autonomously gather information from other nodes.

To organize all this communication, central nodes are required. One of these central nodes, developed within the MARS Framework, is the domain broker. It holds information about the participating nodes and supports the communication between these.

One part of the domain broker is the so called *query broker*. This part enables local and remote nodes to post queries to the Semantic Web, which cannot be answered by a single node, due to the fact that information from multiple domains is required to answer the query.

In this thesis the internal logic for such a query broker is designed and implemented. The implementation contains certain changes to the handling of ontologies and registered domain nodes in the domain broker; as well as the methods of the query broker to handle incoming queries.

Acknowledgments

First of all, I would like to thank Prof. Dr. Wolfgang May for his excellent supervision of this thesis and my whole course of studies.
Furthermore I would like to thank Dr. Franz Schenk for his technical and scientific support and insights and for co-supervising this thesis.
Finally, I would like to thank my sister Prisca Markmann and Jan Gertken for proofreading and my parents for their continued support and encouragement throughout my studies.

Master Thesis

Query-Brokering in Semantic-Web-Umgebungen

Heiko Vollmann

September 29, 2009

supervised by Prof. Dr. Wolfgang May
Databases and Information Systems Group
Institute for Computer Science
University of Göttingen

Contents

1 Introduction 6

2 Basics 10
- 2.1 URI 10
- 2.2 Ontology 11
- 2.3 RDF and RDF Schema 11
  - 2.3.1 RDF 11
  - 2.3.2 RDFS 13
  - 2.3.3 SPARQL 13
- 2.4 OWL 14

3 Software Tools 15
- 3.1 The JENA Framework 15
  - 3.1.1 Jena API 15
- 3.2 Reasoner 16
  - 3.2.1 Pellet Reasoner 16
- 3.3 TDB 17
- 3.4 LOG4J 17

4 Design 18
- 4.1 Example 1 18
- 4.2 Ontology and Domain Nodes 23

4.2.1 LSR . . . 23
4.2.2 Loading the Ontology . . . 23
4.2.3 Registering a Domain Node . . . 25
4.2.4 The Ontology and Domain Node Administration Structure . . . 27
4.3 Hashes . . . 29
4.3.1 Details . . . 29
4.3.2 Special Structures . . . 30
4.3.3 Export . . . 31
4.4 Query Execution . . . 33
4.4.1 Receive Query and Extract Parts . . . 33
4.4.2 Get all Relevant Concepts . . . 34
4.4.3 Get the Supporting Nodes . . . 35
4.4.4 Collect Data from Domain Nodes . . . 35
4.4.5 Execute Query . . . 36
4.4.6 Convert Result and Answer Query . . . 37
4.5 Completeness . . . 38
4.6 Example 2 . . . 40
4.7 Example 3 . . . 43

5 Implementation **46**
5.1 General . . . 46
5.2 Overview . . . 46
5.3 Domain Broker Servlet . . . 47
5.4 Domain Broker Class . . . 48
5.5 The Hash Classes . . . 49
5.6 Query Broker Class . . . 50
5.7 The QueryBox . . . 50
5.8 MARS Frontend . . . 51

6 Conclusion **58**

Bibliography **60**

1 Introduction

This thesis is written as part of the approach of the 'Databases and Information Systems Group' at Göttingen-University and other international research groups, which aim to develop and exemplarily implement a semantic web, as described in [11].

In contrast to the classical web, nodes in the semantic web are capable of communicating with each other. Through this communication, information can be gained which could never be gained by a single node in the classic web.

In the semantic web there are different kinds of nodes, providing different services. In this thesis two kinds of nodes are especially relevant:

- domain nodes: The domain nodes provide data of a certain domain. For example, a domain node could provide political data about countries. This may include the population, birth rate or which cities are located in it.
- domain broker: A domain broker is a central node in the semantic web. It holds meta information about the domain nodes. This includes the knowledge which domains a domain node supports and about which concepts of the domains the domain node can provide data.
  The domain broker also provides certain services for external nodes. For example it provides an interface to execute queries. The actual execution of the query is done by one part of the domain broker called the 'query broker'.

A basic implementation of a domain broker has been done by Tobias Knabke in his thesis 'Entwicklung und Implementierung eines Domain Brokers für das Semantic Web' [1] in 2006.

The focus of [1] was on event and action brokering. The query broker of [1] has the basic input and output methods but very little internal logic. The generation of answers to the incoming queries did not include ontology reasoning. The target of this thesis is to change that.

With the implementation of this thesis, the domain broker should be able to handle multiple ontologies and provide an interface for domain nodes to register themselves at the broker. The query broker should obtain the required data to answer an incoming query from these domain nodes. Due to this construct the domain broker is not required to hold any real data itself.

Figure 1.1 demonstrates how the different components work together.
The ontologies are loaded during startup of the domain broker and are stored by it for later use. When the domain broker is running, the *Interface* is active and the domain nodes can register themselves.
After the domain nodes are registered the domain broker is ready to execute the queries coming from the external nodes. In order to answer theses incoming queries the required data is obtained from the previously registered domain nodes.

A special requirement to the query broker is that the subqueries that are generated and sent to the domain nodes should be constructed in a manner that allows the domain nodes to execute these queries without the use of a reasoner. This is required to speed up the whole process of answering the query, since the use of a reasoner is very time consuming.

A domain node normally provides data for only one domain, or even for only one aspect of a domain. Therefore the query broker needs to be able to query multiple domain nodes.

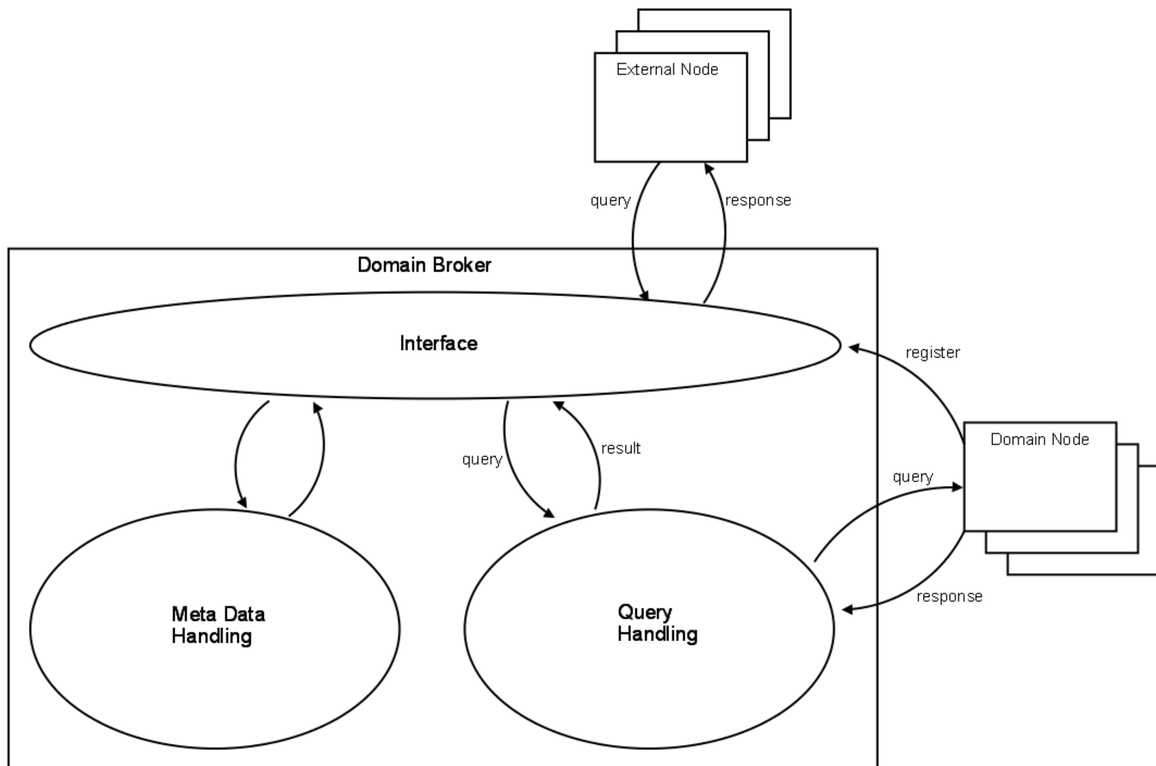


Figure 1.1: The Domain Broker Structure (theoretic)

To meet these requirements, the process of executing a query, done by the query broker, is divided into several steps:

1) Since the subqueries sent to the domain nodes should be executed by those without the use of a reasoner, these subqueries do not only need to contain the concepts occurring in the main query, but also all sub, equivalent and other relevant concepts which would normally be evaluated as important by the reasoner. Because of this, the query broker first needs to evaluate all the classes and properties occurring in the query and find all the additionally required concepts. All these concepts are stored in the 'relevant concepts list'.

2) For each concept in this list the query broker checks which domain nodes support this concept. These correlations are stored.

3) The query broker then generates queries from the concepts and sends them to the correlating domain nodes. These are simple queries that can be answered by domain nodes without a reasoner. The results returned by the domain nodes concern just the ABox (only data about individuals and no meta information). This data is stored by the query broker for later use.

4) When all the data is collected from the domain nodes, the query broker combines it into a new ABox. To this ABox the ontology stored in the domain broker is added as a TBox. On this combination the incoming query is executed. For this final execution a reasoner is used.

In virtue of this partitioning of the query execution, it is possible to fulfill the requirement of generating simple subqueries and still take the meta information required to execute a query into account.

The thesis is structured as follows. The next chapter will give an overview of the concepts and languages used in this thesis. Chapter 3 will then describe the technologies and APIs used to develop and implement the thesis. A basic example of how the new query broker works is given at the beginning of Chapter 4. This example is followed by a detailed description of the design and requirements of the new query broker and the parts that where changed in the domain broker. In Chapter 5 some details on the implementation are described, along with some examples of the new web interface designed to help users and developers. The thesis is concluded with a summary of the important parts designed and implemented.

2 Basics

This thesis is developed and implemented for use in the MARS Framework for Evolution and Reactivity in the Semantic Web, which is introduced in [11].

This chapter will give a basic introduction to the concepts and languages underlying the design and implementation of this thesis. In order to give a complete overview of the concepts some related structures are described that are not used directly in the thesis.

2.1 URI

A Uniform Resource Identifier[1] (URI) is a character string which uniquely identifies a resource. The resource can be abstract, like a website or located in the real world, like a person.
URIs can amongst other be classified as Uniform Resource Locators (URL) or Uniform Resource Names (URN). A URL identifies a resource through its primary access mechanism or its location within the network, for example: 'http://...' directs to a specific webpage. A URN identifies a resource through its name, for example 'isbn:0-19-853737-9' refers to a book, but provides no information where to find or get it.
For more information see [2].

[1] Also called Universal Resource Identifier.

2.2 Ontology

In computer science the term *ontology* specifies a set of definitions which describe the concepts of a domain. The ontology contains information about the hierarchical structure of the classes in the domain, the notation used and the semantics which lie behind the notation. The ontology defines the objects the domain can contain and specifies the relations in which these objects stand to each other. It determines the exact notation to be used in any document about the specified domain.

To signal that a set of notations belongs together and to shorten the actual string representing the notation a namespace can be defined. The namespace defines how a specific part of the URL describing a concept can be replaced by an abbreviation. The part of the URL to be replaced always has to start at the beginning of the URL.

For example it can be defined that *http://www.semwebtech.de/mondial/10/meta#* is the namespace for meta data about the Mondial ontology and that the abbreviation *mon:* can be used to represent it.
With this namespace definition, instead of *http://www.semwebtech.de/mondial/10/meta#hasCity* the shorter string *mon:hasCity* can be used.

A collection of information is divided into two sections. The statements building the ontology are located in the so called TBox. The statements describing individuals are located in the ABox.
For more information see [13].

2.3 RDF and RDF Schema

2.3.1 RDF

The Resource Description Framework (RDF) is a structure to describe resources and denote this description in a machine readable and machine processable way.
RDF uses URIs (see above) to uniquely identify a resource. Because of this, the resource

can be located anywhere within the net or the real world. Furthermore, one resource can be described by many RDF models located in different places within the net. Through the use of URIs, the distributed information about one resource can be brought together and the user can tell that the different RDF models refer to the same resource.

The RDF Model is composed of Triplets. The Triplets contain:

- a *subject* which always is the URI of the resource, this triple gives information about,
- a *predicate* which names the aspect of the resource to be described in this triple,
- an *object* which is the value of the description. This can be a literal (string or number) or another resource.

An RDF Model can be represented for example as a graph, as a list of triples or as a special XML markup called RDF/XML.
The following is an example which illustrates the basic idea of what an RDF triple could look like in N-triplets:

```
<http://www.semwebtech.de/mondial/10/countries/D>
   <http://www.semwebtech.de/mondial/10/meta#hasCity>
       <http://www.semwebtech.de/mondial/10/cities/Goettingen> .
<http://www.semwebtech.de/mondial/10/countries/D>
  <http://www.semwebtech.de/mondial/10/meta#name> "Germany" .
```

This gives a description about a resource which can be accessed by the URI http://www.myworld.de/country/D. It contains the information that this resource has a *hasCity* relation to another resource identified by http://www.myworld.de/city/Goettingen and a *name* property which is the literal 'Germany'.

The same example denoted in N3 looks like this:

```
@prefix mon: <http://www.semwebtech.de/mondial/10/meta#>.
<http://www.semwebtech.de/mondial/10/countries/D>
```

```
<mon:hasCity> <http://www.semwebtech.de/mondial/10/cities/Goettingen> ;
<mon:name> "Germany" .
```

For more information see [4].

2.3.2 RDFS

RDF Schema (RDFS) is a vocabulary description language.
Via an RDF Schema the vocabulary of a specific domain is defined. A Schema provides meta data to the data contained in the RDF model. The meta data can, for example, contain class declarations and information about the hierarchy of the classes (*rdfs:subClassOf*). Furthermore it can contain domain and range information about the properties used in the RDF model. Simple ontologies can be represented in RDFS.
Syntactically an RDFS document is a valid RDF Model and can be read and processed with the same mechanisms as RDF.
For more information see [5].

2.3.3 SPARQL

SPARQL is the recursive acronym that stands for SPARQL Protocol And RDF Query Language.
SPARQL is a query language used to access RDF graphs. The SPARQL syntax is based on known SQL commands, but also allows variable bindings and optional parameters.
To make the queries easier to read and write for humans, SPARQL provides methods for namespace handling: At the beginning of a query the 'PREFIX' keyword can be given to define a set of namespaces for the query. The query itself can then use briefer names, called *local names*.
In order to understand the basic idea, consider the following short query, which illustrates what a SPARQL query could look like:

```
PREFIX mon: <http://www.semwebtech.de/mondial/10/meta#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?names
   {
   ?country rdf:type mon:Country.
   ?country mon:name ?names
   }
```

Explanation: The query will search the RDF graph for all resources which have a property 'rdf:type' and 'mon:Country' as a value of this property. For all these resources it selects the value of the 'mon:name' property and returns these values in the names field.
For more information see [6].

Since the expressive power of RDFS is not sufficient to express all ontology features needed in the Semantic Web and since this thesis is developed to be used in the Semantic Web, another language is needed. This is described in the next chapter.

2.4 OWL

The Web Ontology Language (OWL) is a language used to define and instantiate ontologies. It is technically based on RDF syntax, but extends RDF and RDFS in making it possible to express semantics related to the data.
OWL is designed to provide a notation for the structure and semantics defined by the theoretical construct of an ontology needed for the Semantic Web.
This is a small example of a relation needed for reasoning that cannot be expressed in RDFS: *hasCity* is the inverse of *cityIn*.
In OWL syntax it looks like this: *mon:hasCity owl:inverseOf mon:cityIn.*
For more information see [3].

3 Software Tools

3.1 The JENA Framework

Jena is an open source framework developed to provide structures and methods needed to build Semantic Web applications. It has developed out of work on the HP Labs Semantic Web Programme [10]. It provides methods to handle RDF, RDFS, OWL and SPARQL.

3.1.1 Jena API

The Jena API provides methods to handle RDF data.
The main purpose of the Jena API is to store and access an RDF model with Java Classes. It provides methods to load RDF data from one or more files and builds an RDF model from the data. This model can then be accessed and altered through numerous API functions. Simple but limited query and navigation methods are provided for the RDF graph, as well as export functions (for example saving the graph in an RDF/XML markup).

Since the built-in navigation function is limited to simple queries, the Jena API provides an engine to process queries written in common query languages, such as SPARQL. This engine is called ARQ[1].

[1] Since ARQ is not important for this thesis, no further details are given.

3.2 Reasoner

By the use of a reasoner, an API can draw conclusions about the RDF graph. The meta information can be given in RDFS or OWL code terms within the model. All queries posed or commands given to the RDF model will be processed by the reasoner.

The Jena API also provides methods for loading a reasoner. The reasoner can either be one of those provided by the API or it can be an external one, as for example the 'Pellet Reasoner' (see below).

For the purpose of illustration, consider this example:
Assume that in the RDF model there are *Country* and *City* objects. Each *City* has a *mon:cityIn* property which directs to the *Country* where the *City* is located. The *Country* has no information about what *City* lies in it.

Assume further that the OWL meta data in the RDF model contains the following entry: mon:hasCity owl:inverseOf mon:cityIn. Now consider sending the following SPARQL request to the model:

```
SELECT ?city
WHERE {<http://www.myworld.de/countries/D/> mon:hasCity ?city}
```

Because the reasoner applies the information that *mon:hasCity* is the inverse of *mon:cityIn* to the model, the answer of this query will be all cities that are located in the country http://www.myworld.de/countries/D/. Without the reasoning the answer would be empty, since the *Country* objects do not actually have a property *mon:hasCity*.

3.2.1 Pellet Reasoner

The Pellet reasoner was originally developed at the University of Maryland's Mindswap Lab. It is designed to provide the full expressiveness of OWL-DL for Java applications.
The reasoner can be integrated directly into the Jena model creation process.
For more information see [9].

3.3 TDB

TDB is a component of the Jena Framework. It provides methods to permanently store RDF data on a single machine.

All data representing one model is stored in one folder given by the user. The data is stored in an RDF native format. Therefore no external relational database is necessary.
To store the data in the TDB database a connection between an RDF model and a TDB storage is created. The model can then be used like any other RDF model in Jena.
The TDB storage can be accessed via a command line utility or via methods provided for the Jena API. (In this thesis the API is used to access the database.)
For more information see [7].

3.4 LOG4J

Log4j is a framework for logging messages in Java applications. It is part of the 'Apache Software Foundation'. Its main purpose is to provide a simple-to-use method for logging during development, debugging and deployment of an application.

Log4j provides six basic logging levels: FATAL, ERROR, WARN, INFO, DEBUG and TRACE. Theses logging levels are hierarchical. In this case this means that, if the log level is set to WARN all WARN, ERROR and FATAL messages will be logged.

The configuration for Log4j is given in a separate file[2]. Therefore it is not necessary to edit the source code in order to change the log level, the output device (stdout, file, etc.) or the logging format.
For more information see [8].

[2]Filename: log4j.properties

4 Design

In this Chapter, the design of the new query broker is presented.
The first section provides a basic example. It demonstrates what happens during the domain broker startup and how the query broker handles an incoming query.
The following sections describe the procedures in detail.
The last two sections provide examples, which illustrate the use of a special structure.

4.1 Example 1

In order to acquire a better understanding of how the query broker works and what is required for it to function, an example is given in what follows. In this example only the procedure is described. The exact details are given later in this chapter.

During the startup the domain broker acquires information about which domains it should support, obtains the corresponding ontologies and creates its internal ontology with these. Let us assume that the domain broker supports the Mondial domain and therefore loads the Mondial ontology [15].

After the domain broker has been started, two domain nodes register themselves at the domain broker and inform it that they support the following concepts from the Mondial ontology:

1. Mondial politics: This domain node provides data about countries, provinces and cities. Among other things this includes their names, correlation and other political information.

2. Mondial organizations: This domain node provides data about organizations. This includes which organizations exist, their names, members, headquarters, etc.

At this point the domain broker is capable of answering queries about the Mondial domain. Now let us assume an external node sends a request to the domain broker with a SPARQL query looking like this:

```
PREFIX mon:<http://www.semwebtech.org/mondial/10/meta#>
SELECT ?on ?hn
WHERE {
  ?o a mon:Organization .
  ?o mon:abbrev ?on .
  ?o mon:hasHeadq ?h .
  ?o mon:hasMember ?c .
  ?c mon:capital ?h .
  ?h mon:name ?hn
}
```

The domain broker passes this query on to the query broker to execute it and to generate an answer.
The query broker first extracts and expands all the concepts within the query, which are these:

```
http://www.semwebtech.org/mondial/10/meta#Organization
http://www.semwebtech.org/mondial/10/meta#abbrev
http://www.semwebtech.org/mondial/10/meta#hasHeadq
http://www.semwebtech.org/mondial/10/meta#hasMember
http://www.semwebtech.org/mondial/10/meta#capital
http://www.semwebtech.org/mondial/10/meta#name
http://www.semwebtech.org/mondial/10/meta#isMember
http://www.semwebtech.org/mondial/10/meta#isCapitalOf
```

Here it can be seen that the *mon:isMember* and *mon:isCapitalOf* concepts are added to the list of concepts occurring in the query. This is due to the fact that they are inverse to concepts

in the query and therefore relevant to the execution of the query.

The query broker then checks which domain nodes support which concept. In this example the following is the case:

```
http://www.semwebtech.org/nodes/2007/mondialPolitic {
        http://www.semwebtech.org/mondial/10/meta#capital
        http://www.semwebtech.org/mondial/10/meta#name
        http://www.semwebtech.org/mondial/10/meta#isCapitalOf
}
http://www.semwebtech.org/nodes/2007/mondialOrganization {
        http://www.semwebtech.org/mondial/10/meta#Organization
        http://www.semwebtech.org/mondial/10/meta#abbrev
        http://www.semwebtech.org/mondial/10/meta#hasHeadq
        http://www.semwebtech.org/mondial/10/meta#hasMember
        http://www.semwebtech.org/mondial/10/meta#name
        http://www.semwebtech.org/mondial/10/meta#isMember
}
```

In this notation, the list of domain nodes which support some of the relevant concepts can be seen. In this case these are the *.../mondialPolitic* and *.../mondialOrganization* node. Within the cambered brackets a list of concepts supported by the node preceding the brackets is given.

From each correlation between a domain node and a concept one query is created. This query is then send to the domain node. The answer from the domain node contains all data that it has according to the concept in question. The answers to all these queries are combined and stored by the domain broker.
This is a small sample of the answers stored by the query broker:

```
<http://www.semwebtech.org/mondial/10/countries/L/>
  <http://www.semwebtech.org/mondial/10/meta#isMember>
    <http://www.semwebtech.org/mondial/10/organizations/NATO/>
<http://www.semwebtech.org/mondial/10/organizations/G-10/>
  <http://www.semwebtech.org/mondial/10/meta#abbrev>
```

```
    "G-10"
<http://www.semwebtech.org/mondial/10/countries/I/>
  <http://www.semwebtech.org/mondial/10/meta#capital>
    <http://www.semwebtech.org/mondial/10/countries/I/provinces/Lazio/cities/Rome/>
<http://www.semwebtech.org/mondial/10/countries/I/>
  <http://www.semwebtech.org/mondial/10/meta#name>
    "Italy"
```

When all the data is collected from the domain nodes, the query broker uses the data combined with the ontology stored by the domain broker to execute the main query. The result is then returned to the domain broker.
The domain broker converts this result into a MARS conform structure, which looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<logic:variable-bindings xmlns:logic="http://www.semwebtech.org/lang/2006/logic"
    xmlns:xs="http://www.w3.org/2001/XMLSchema#" xmlns:rdf="http://www.w3.org
    /1999/02/22-rdf-syntax-ns#" xmlns:sr="http://www.w3.org/2005/sparql-results#">
  <logic:tuple>
    <logic:variable name="on">
      <literal xmlns="http://www.w3.org/2005/sparql-results#">Inmarsat</literal>
    </logic:variable>
    <logic:variable name="hn">
      <literal xmlns="http://www.w3.org/2005/sparql-results#">London</literal>
    </logic:variable>
  </logic:tuple>
  <logic:tuple>
    <logic:variable name="on">
      <literal xmlns="http://www.w3.org/2005/sparql-results#">NIB</literal>
    </logic:variable>
    <logic:variable name="hn">
      <literal xmlns="http://www.w3.org/2005/sparql-results#">Helsinki</literal>
    </logic:variable>
```

```
  </logic:tuple>
  [...]
  <logic:tuple>
    <logic:variable name="on">
      <literal xmlns="http://www.w3.org/2005/sparql-results#">Benelux</literal>
    </logic:variable>
    <logic:variable name="hn">
      <literal xmlns="http://www.w3.org/2005/sparql-results#">Brussels</literal>
    </logic:variable>
  </logic:tuple>
</logic:variable-bindings>
```

The formatted result is then sent back to the requesting external node.

4.2 Ontology and Domain Nodes

During startup and initialization of the application the domain broker needs to obtain the ontologies of all domains it provides services for. To load the ontologies, the domain broker requires the LSR (Languages and Services Registry). The LSR provides all the necessary information about all the nodes in the Semantic Web. This includes the domains the domain broker should support and information about how to acquire the ontologies to these domains. Furthermore the LSR provides information about the domain nodes. This information is required later on during the domain registration.

4.2.1 LSR

The present development state of the MARS Framework does not provide a web service for the LSR. Therefore the LSR is represented by a large XML file. This file contains all information the LSR will provide later on.
In this thesis the LSR file is read and parsed into a *jdom.Document*. The document is then queried to get the necessary information required by the domain broker. (for details see [14])

4.2.2 Loading the Ontology

The LSR contains information about the configuration of the domain broker. The domain broker obtains this information and configures itself with it. How exactly this takes place is described in this section.
The configuration information is represented by an XML document, which could look like this:

```
<mars:DomainBroker
   rdf:about="http://localhost:8080/services/2007/domain-broker"
   xml:base="http://www.semwebtech.org/services/2007/domain-broker/">
  <mars:name>All Domains Broker</mars:name>
```

```
 <has-task-description>
  <TaskDescription>
   <describes-task rdf:resource="&mars;/domain-broker#register-for-event"/>
   <provided-at rdf:resource="register-for-event"/>
   <Reply-To>body</Reply-To>
   <Subject>n.a.</Subject>
   <input>element register</input>
   <variables>no</variables>
  </TaskDescription>
 </has-task-description>
 [...]
 <mars:supports-domain rdf:resource="http://www.semwebtech.org/domains/2006/
     travel#"/>
 <mars:supports-domain rdf:resource="http://www.semwebtech.org/domains/2006/
     mondial#"/>
</mars:DomainBroker>
```

Within the <mars:supports-domain> elements the URIs of the supported domains are provided.

At this point the domain broker 'knows' which domains it supports.
To obtain the URIs of the ontologies corresponding to theses domains, the domain broker then requests the domain information from the LSR. As an answer to this request the LSR provides the domain broker with an XML document, containing the domain information. This is an example how this document may look like:

```
<mars:Domain rdf:about="http://www.semwebtech.org/domains/2006/mondial#">
 <mars:name>Mondial Database</mars:name>
 <mars:shortname>mondial</mars:shortname>
 <mars:hasOntology rdf:resource="http://www.someServer.de/mondial-meta.rdf"/>
</mars:Domain>
```

The <mars:hasOntology> element provides information about where to find the ontology corresponding to the described domain.

The domain broker then creates one TDB instance per domain. The ontology file is loaded from the previously obtained URL and stored in the TDB. The location of the TDB instance is stored in the 'ontology and domain node administration structure' (see below for description).

After all the ontologies have been stored in the TDB instances, the domain broker collects all the ontologies and combines them into one ontology. This combined ontology is then stored within the *Ontology* object in the domain broker. This *Ontology* object is used during the query execution to obtain the necessary meta information required to handle the query.

4.2.3 Registering a Domain Node

The domain broker provides an interface for domain nodes to register themselves at the domain broker. A domain node can use the *http://URL of DomainBroker.de/register-domainNode* URL to register at the domain broker. It needs to provide its own URL to identify itself.
As mentioned before, the LSR provides information about all known domain nodes within the MARS network. To register the domain node the domain broker first obtains this information from the LSR. The LSR provides the information in the form of an XML document. This is an example of how this XML document may look like:

```
<mars:DomainService rdf:about="http://www.semwebtech.org/nodes/2007/monPolitic">
 <mars:name>Mondial Datenbank ueber Politik</mars:name>
 <mars:uses-domain rdf:resource="http://www.semwebtech.org/domains/2006/mondial"/>
 <mars:supports rdf:resource="http://www.semwebtech.org/mondial/10/meta#Country"/>
 <mars:supports rdf:resource="http://www.semwebtech.org/mondial/10/meta#City"/>
 <mars:supports rdf:resource="http://www.semwebtech.org/mondial/10/meta#hasCity"/>
 <mars:supports rdf:resource="http://www.semwebtech.org/mondial/10/meta#area"/>
</mars:DomainService>
```

The domain broker requires information about the concepts the domain node supports. These are found within the <mars:supports> elements of the XML document.
For each domain node being registered, the domain broker creates one TDB instance. To be able to access it later on, an entry in the 'ontology and domain node administration structure' is created for the TDB instance, similar to the ones created for the ontologies.
To store the information about the supported concepts in the TDB, the domain broker first determines if the concept represents a Class or a Property. To do so, the previously created ontology is used. A triplet is then created for each concept and stored in the TDB instance. The triples are created from the following parts:

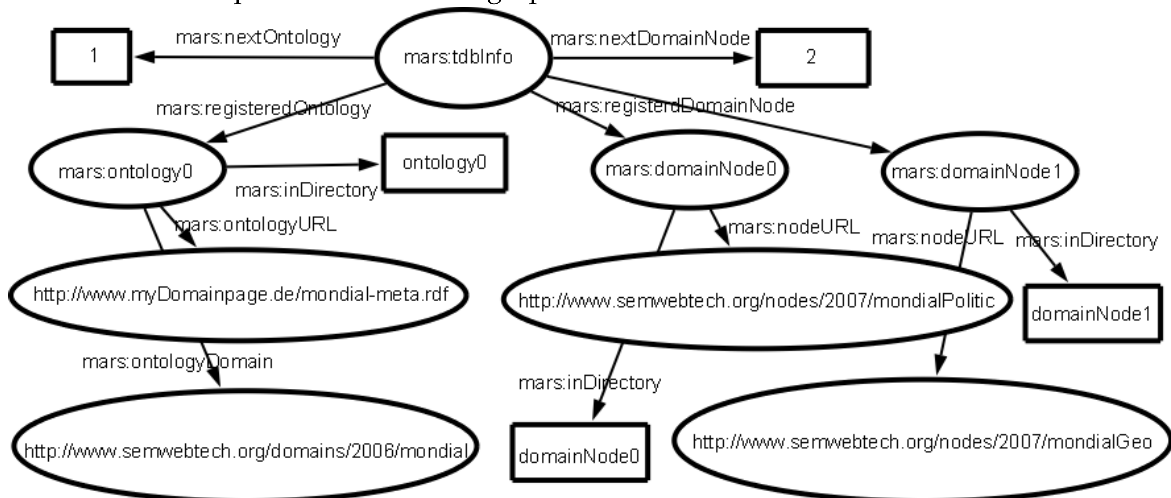
- Subject: The URL of the domain node that supports this concept, which is given by the domain node itself as mentioned above.
- Property: If the concept represents a Class, this Property is <mars:supportsClass>. Otherwise the concept represents a Property and this Property is <mars:supportsProperty>.
- Object: The concept in question.

The following is an example of how a domain registration TDB could look like:

4.2.4 The Ontology and Domain Node Administration Structure

The domain broker needs to be able to change the information about its ontologies and registered domain nodes during runtime. To be able to do so, the domain broker stores the information about these in separate TDB's as mentioned in the Sections above. To access and change the stored information an administration structure is required, which handles the information where the TDB's are stored.
The administration structure itself is stored in a TDB, too. Therefore the required information can be accessed easily through SPAQL requests.

This is an example of how the RDF graph of the administration structure could look like:



The <mars:tdbInfo> node represents the master node.
The <mars:nextOntology> property connects the <mars:tdbInfo> to an integer value. This value represents the number under which the domain broker should save the next incoming ontology.
The <mars:nextDomainNode> works the same way for domain nodes as <mars:nextOntology> does for ontologies.

A <mars:ontologyX> node represents an ontology saved by the domain broker. The 'X' stands for the sequential number under which the ontology is saved. The <mars:ontologyX> node provides some information about the saved ontology. This information is connected to the node through the following properties.

- <mars:ontologyDomain> provides the URI of the domain, which acts as the identifier for the domain.
- <mars:ontologyURL> provides the URL of the ontology file. This is where the ontology to the domain can be found.
- <mars:inDirectory> provides the location of the TDB where the ontology is saved. The literal does not provide the complete URI but only the name of the directory within the basic ontology directory.

A <mars:domainNode> node represents a domain node registered by the domain broker. It works the same way as a <mars:ontologyX>.
The properties provided by the node are:

- <mars:nodeURL> provides the URL to the domain node. This is the address which the query execution sends the generated subqueries to.
- <mars:inDirectory> provides the location of the TDB where the domain node information is saved.

4.3 Hashes

During startup and initialization of the application two hashes are created. One for classes and one for properties. They are generated by a Factory Class, which provides the necessary methods to build the hashes. The Factory Class uses the previously loaded ontology and the information about the domain nodes, which are both stored in the TDB databases at this point.

During the query execution all relevant concepts within the query in question are extracted. (See Section 4.4 for details.) Based on these concepts the subqueries are created. To create the subqueries, which are sent to the domain nodes, two important pieces of information have to be provided for each concept:

1. What other concepts are relevant to evaluate the concept in question without a reasoner?
2. Which domain node supports which concept?

The hashes provide a quick and comfortable way to access this information about a concept.

4.3.1 Details

The HashFactory Class creates two hashes. One hash holds all the necessary information about all classes described in the ontology. The other hash does the same for all properties.

Both hashes have the same basic structure, which is as follows:
All classes, properties and domain node URLs are stored as a *Jena.Resource*. Since a *Jena.Resource* contains both the namespace and the given name of a class or property it is not necessary to handle them explicitly.
The hashes use the concepts from the ontology as keys and associate them with three lists:

- usedClasses
  The *usedClasses* list contains all classes required to expand the concept in question.

This list may appear in both hashes, since a conclusion can be drawn from a property to class through a <owl:Restriction>.

- usesProperty
  The *usesProperty* list contains all properties required to expand the concept in question. This list appears in both hashes. Because of the structure of an <owl:Restriction>, a class may require a property for its evaluation. (see below)

- supportedBy
  The *supportedBy* list contains the URIs of all domain nodes that support the concept in question.

The query broker can request each list directly during its evaluation of a concept. The whole list containing the *Jena.Resource*s is returned to such a request.

4.3.2 Special Structures

The OWL language contains some special structures to define an *owl:Class*. Two of these require a special handling within the hashes:

- owl:intersectionOf
  An *owl:intersectionOf* defines the construction of a new class as the intersection of two or more existing classes.
  The new class is an *owl:subClassOf* of the existing classes. Therefore, after the insertion of the *owl:subClassOf* and the *owl:intersectionOf* structures into the hash, an *usesClass* loop exists. The new Class uses the existing classes via the *owl:intersectionOf* and the existing classes use the new class via the *owl:subClassOf*. To solve this loop, the new class is removed from the *usesClass* list of all existing classes. This results in the correct behavior, since the new class is not needed to evaluate one of the existing classes.

- owl:Restriction
  An *owl:Restriction* defines the construction of a new class through a property and one or more existing classes as objects to the given property. Since it is not necessary for the property and the existing classes to be supported by the same domain node, they are stored separately. The bond between the property and the classes is dissolved, so that the query broker can send both of them to different domain nodes. (for details see 4.4)

4.3.3 Export

The hashes provide a method to export themselves as a string representing an XML Document.

The root element of the XML Document states what this document is about (*propertyHash* or *classHash*) and provides all necessary namespaces.

The child elements of the root element represent the concepts about which information is provided. The name of the element declares what kind of concept is described. (*aboutProperty* or *aboutClass*) The URI of the property or class is provided within the *rdf:resource* attribute.

The 'about a concept' elements can have three kinds of children:

- usesProperty
- usesClass
- supportedBy

All three provide the URI's of the concept or domain node in an *rdf:resource* attribute.

Example:

```
<ClassHash xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 <aboutClass rdf:resource="http://www.semwebtech.org/mondial/10/meta#unionCC">
  <usesClass rdf:resource="http://www.semwebtech.org/mondial/10/meta#City" />
  <usesClass rdf:resource="http://www.semwebtech.org/mondial/10/meta#Country" />
```

```
</aboutClass>
<aboutClass rdf:resource="http://www.semwebtech.org/mondial/10/meta#Country">
 <supportedBy rdf:resource="http://www.semwebtech.org/nodes/2007/mondialPolitic"/
     >
 </aboutClass>
</ClassHash>
```

4.4 Query Execution

In order to process an incoming query and generate an answer to that query several steps are required:

1. Receive the query and extract the important parts from it.
2. Get all the relevant concepts necessary to execute the query.
3. Get the supporting domain nodes to these relevant concepts.
4. Query the domain nodes for the concepts and collect the data they can provide about these concepts.
5. Execute the query on the collected data to get the final result.
6. Convert the result into a MARS conform form and send it to the querying web service.

Steps two to five are handled by the query broker. Step one and six are handled by the domain broker itself.

4.4.1 Receive Query and Extract Parts

The domain broker provides an interface to execute a query. This interface can be accessed by sending a string representing an XML document to the URI http://www.domain_broker.de/execute-query. The XML document contains specific elements, which will now be explained.
This is an example of what such an XML document could look like:

```
<query>
  <opaque language="http://www.w3.org/2005/01/sparql-protocol#">
    PREFIX travel:&lt;http://www.semwebtech.org/domains/2006/travel#&gt;
    PREFIX mon:&lt;http://www.semwebtech.org/mondial/10/meta#&gt;
```

```
    SELECT ?a ?b
     WHERE {
       ?a travel:from ?b.
       ?b mon:name 'Bremen'.
     }
  </opaque>
  <reply-to>http://www.external_node.de/receive-query-answer</reply-to>
</query>
```

The <opaque> element has a *language* property describing what language the query is written in. (In this thesis the SPARQL query broker is the only one implemented.)
The content of the <opaque> element is the query that should be executed.
The <reply-to> element contains the URL where the answer to the query should be sent to by the domain broker.

To execute the query the domain broker first deduces what language the query is written in and therefore which query broker is required to handle it. (This is not necessary here, since there is only one supported language.) It then creates and starts a new thread to handle the query. Within the thread the previously determined query broker is provided with the query and instructed to generate a result.

4.4.2 Get all Relevant Concepts

The query broker generates a list where all the relevant concepts are collected. To start the list, all concepts from the query are added. After this, all concepts newly added to the list are recursively processed. In each recursion the domain broker gets all required concepts from the class hash and the property hash (see Section 4.3). In addition to this, the rules given by the MARS framework are evaluated and the concepts provided by these rules are added to the list. (The rule evaluation is part of the previously existing query broker described in [1] and not part of this thesis.)

4.4.3 Get the Supporting Nodes

After the list of relevant concepts is created, the query broker needs to find out which domain node can provide data to which relevant concept in order to generate the subqueries. This information is provided by the class hash and the property hash (see Section 4.3).
To store the information about correlations between domain nodes and relevant concepts the query broker creates a hash. The keys of this hash represent the URLs of the domain nodes that can provide data. Assigned to each URL is a list of the concepts that are supported by the domain node.

The following is a visualization of the 'supporting domain nodes hash' to get a better understanding of it:

```
http://www.semwebtech.org/nodes/2007/mondialPolitic {
       http://www.semwebtech.org/mondial/10/meta#name
       http://www.semwebtech.org/mondial/10/meta#City
}
http://www.semwebtech.org/nodes/2007/travel-bahn {
       http://www.semwebtech.org/domains/2006/travel#from
}
```

4.4.4 Collect Data from Domain Nodes

For each concept in the supporting domain nodes hash a query is generated and sent to the domain nodes which supports the concept.

Up to this point, all concepts are handled regardless of which type they really are. Since the generated subquery depends upon whether the concept is a class or a property, the query broker needs to find out of what type the concept is. This is done by a query to the ontology of the domain broker. If the concept represents a class X, the following query is generated and send to the domain node.

```
SELECT ?a ?b
```

```
    WHERE {
    { ?a ?b <X> }
  }
```

If the concept represents a property Y, the query sent to the domain node looks like this:

```
SELECT ?a ?b
    WHERE {
        ?a <Y> ?b
      }
```

For each result returned by the domain node a triple is then created. The triple is build from the relevant parts of the ABox returned by the domain node. In the case of a class if looks like this: *?a ?b* <*X*>. If the concept represents a property it looks like this: *?a* <*Y*> *?b*. The query broker creates a TDB for the main query. The directory of the TDB is given a name based on the number of the query.

All the triples generated from the results to the subqueries are then inserted into this TDB.

Since it may take a while to receive the answers from the domain nodes, this whole procedure may take some time. This is the main reason why the whole query execution is running within a thread, so that the query broker is capable of doing other things while waiting for the answers from the domain nodes.

4.4.5 Execute Query

As soon as all the subqueries are executed and all the results are saved into the TDB, the query broker can process the main query, since at this point all necessary data is available (see Section 4.5 for illustration).

To execute the query the query broker builds a model from the TDB and adds to this model the ontology from the domain broker. The query is then executed on this model using a reasoner. The answer to this query is the final result which should be send to the reply-to

address.

4.4.6 Convert Result and Answer Query

The final result is returned by the query broker to the domain broker.
The domain broker converts the result into a MARS conform markup. The methods for this conversion have been created in [1] and have not been changed in this thesis. The correctly formatted result is then sent back to the web server which sent the request.
This is an example of what the final result in the correct markup could look like:

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="a"/>
    <variable name="b"/>
  </head>
  <results>
    <result>
      <binding name="a">
        <uri>http://www.semwebtech.org/domains/2006/travel#Bremen_Hbf-Oldenburg_Hbf
            </uri>
      </binding>
      <binding name="b">
        <uri>http://www.semwebtech.org/mondial/10/countries/D/provinces/Bremen/
            cities/Bremen/</uri>
      </binding>
    </result>
  </results>
</sparql>
```

4.5 Completeness

In this Section a short explanation is given of why the design of the query execution produces correct and complete answers.

To execute a query and generate a correct answer to that query a model is required that contains all the information necessary to execute that query. Depending on the query the requirements on the model variate. If the query can be executed correctly without a reasoner only an ABox is required. This ABox needs to hold all the information required for that particular query. If, on the other hand, reasoning is required to execute the query correctly, a TBox is required in addition to the ABox. This TBox needs to contain all the meta information required to execute the query.

The following will demonstrate that the design given above provides a model containing all the information required to execute the query. To be more precise, the query broker provides a model containing all the information it can provide to execute the query. This may not be enough to really execute the query correctly. For example, if the query contains concepts from domains unknown to the domain broker, it is not capable of answering this query correctly.

ABox completeness

The ABox of the model on which the incoming query is executed is created from all the answers to the subqueries. Two things are required for the ABox to be complete.

The first is that all domain nodes that may provide relevant data to the query execution should be queried. This requirement is fulfilled through the use of the 'supporting domain nodes hash' which contains all the information the query broker has about which domain nodes provide data to the concepts in the query.

The second requirement is that the queried domain nodes should provide all the data known to them about the concepts in question. This requirement is fulfilled because of the method according to which the subqueries are created. The subqueries are very simple and contain only one concept. Due to this design the only possible answer the domain nodes

can give is the part of their ABox, containing individuals to that concept.

Since these two requirements are met and the final ABox is constructed from the ABoxes got from the domain nodes, the ABox contains all the information that is required to execute the query, or at least all the information the query broker can possibly provide.

TBox completeness

The TBox of the model on which the incoming query is executed is created from the ontology stored within the domain broker. This ontology is a combination of all the ontologies from all the domains supported by the domain broker. Therefore it holds all meta information from all the domains the query broker is capable of answering queries to.

Since the ABox and TBox are complete, or as complete as the query broker can build them given the information it has, the final query execution generates the best possible answer.

Normally the domain broker only receives queries about domains it supports and it knows enough domain nodes to get data from. In this case (the usual case) the answers generated by the query broker are correct.

4.6 Example 2

In this chapter some examples are given to further illustrate how to use the new domain broker and to demonstrate its capabilities.

The following example is given to show that the new query broker is capable of handling the OPTIONAL and FILTER constructs of SPARQL. In addition, it shows that sometimes more than only the required data is collected.

Let's assume for the sake of the example that like in Example 1 the Mondial ontology has been loaded by the domain broker and that the two domain nodes given in Example 1 registered themselves at the domain broker. In addition to this, the following third domain node registered itself:

3) Mondial geo: This domain node provides data about geographical structures like rivers, islands, mountains, etc. It provides their names, coordinates and locations.

To this setup an external node sends the following query:

```
<query>
  <opaque language="http://www.w3.org/2005/01/sparql-protocol#">
    PREFIX mon:&lt;http://www.semwebtech.org/mondial/10/meta#&gt;
      SELECT ?c
       WHERE {
          ?c a mon:Country .
           optional { ?b a mon:Mountain. ?b mon:locatedIn ?c. }
           FILTER (!bound(?b))
       }
  </opaque>
  <reply-to>http://www.external_node.de/receive-query-answer</reply-to>
</query>
```

As a result to this request a list is expected, containing all the countries which do not contain a mountain. The query broker gets this query from the domain broker and first

extracts and expands all concepts. It then checks which of the registered domain nodes support any of these concepts and generates the 'supporting domain nodes hash' which in this case looks like this:

```
http://www.semwebtech.org/nodes/2007/mondialPolitic {
        http://www.semwebtech.org/mondial/10/meta#Country
}

http://www.semwebtech.org/nodes/2007/mondialGeo {
        http://www.semwebtech.org/mondial/10/meta#Mountain
        http://www.semwebtech.org/mondial/10/meta#locatedIn
}
```

The query broker will then collect all the relevant data from the domain nodes.
Note that in this case it will collect more than only the required data. A human looking at and understanding the query and the structure of the domain nodes can see that it is enough to get the data from the 'mondial geo' domain node, since this domain node will return all the countries, mountains and locations. The 'mondial politics' domain node will not contribute any significant data. But since the query broker has no means to determine this information prior to executing the main query, it has to collect all the data that might contribute information to the final query execution.

The query broker collects data to all concepts. The fact that there are special structures within the query (in this case both OPTIONAL and FILTER) is not taken into account at this point. Therefore all necessary data is collected by the query broker.

After collecting all the data, the query broker executes the main query. Since it is the original query, sent by the external node, which is being executed at this point, the special structures are taken into account and the correct answer is returned. This answer looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<logic:variable-bindings xmlns:logic="http://www.semwebtech.org/lang/2006/logic"
    xmlns:xs="http://www.w3.org/2001/XMLSchema#" xmlns:rdf="http://www.w3.org
```

```
    /1999/02/22-rdf-syntax-ns#" xmlns:sr="http://www.w3.org/2005/sparql-results#">
  <logic:tuple>
    <logic:variable name="c">
      <uri xmlns="http://www.w3.org/2005/sparql-results#">http://www.semwebtech.org
          /mondial/10/countries/LT/</uri>
    </logic:variable>
  </logic:tuple>
  <logic:tuple>
    <logic:variable name="c">
      <uri xmlns="http://www.w3.org/2005/sparql-results#">http://www.semwebtech.org
          /mondial/10/countries/MC/</uri>
    </logic:variable>
  </logic:tuple>
[...]
</logic:variable-bindings>
```

4.7 Example 3

In this example the handling of a restriction is demonstrated.

As in Examples 1 and 2 the Mondial ontology is used in this example too. The domain node 1) is the only one necessary for this example.

Since there are no restrictions in the original Mondial ontology, there is one added for this example. The following additions have been made to the Mondial ontology:

```
<rdf:Description rdf:about="http://www.semwebtech.org/mondial/10/meta#myNeighbors"
    >
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
    <owl:equivalentClass>
       <owl:Restriction>
         <owl:onProperty rdf:resource="http://www.semwebtech.org/mondial/10/meta#
             neighbor" />
         <owl:someValuesFrom rdf:resource="http://www.semwebtech.org/mondial/10/
             meta#myCountry" />
      </owl:Restriction>
    </owl:equivalentClass>
  </rdf:Description>
 <rdf:Description rdf:about="http://www.semwebtech.org/mondial/10/meta#myCountry">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  </rdf:Description>
```

Furthermore in the data base of domain node 1) the object representing the country 'Germany' was changed to be of the type <mon:myCountry>.

```
<rdf:Description rdf:about="http://www.semwebtech.org/mondial/10/countries/D/">
   <rdf:type rdf:resource="http://www.semwebtech.org/mondial/10/meta#myCountry"/>
   <name>Germany</name>
   [...]
</rdf:Description>
```

After these changes, the domain broker is started and the domain node is registered. Now the following query is sent to the domain broker:

```
<query>
  <opaque language="http://www.w3.org/2005/01/sparql-protocol#">
    PREFIX mon:&lt;http://www.semwebtech.org/mondial/10/meta#&gt;
      SELECT ?neighbors
       WHERE {
         ?neighbors a mon:myNeighbors
       }
  </opaque>
  <reply-to>http://www.external_node.de/receive-query-answer</reply-to>
</query>
```

As always, the domain broker passes the query on to the query broker, which first extracts and expands the concepts. The expanded concept list looks like this:

```
http://www.semwebtech.org/mondial/10/meta#myNeighbors
http://www.semwebtech.org/mondial/10/meta#myCountry
http://www.semwebtech.org/mondial/10/meta#neighbor
```

It can be seen here that the domain broker added the class <mon:myCountry> and the property <mon:neighbor>. These are required to resolve the restriction <mon:myNeighbors>. Now the query broker builds the 'supporting domain nodes hash', which in this case looks like this:

```
http://www.semwebtech.org/nodes/2007/mondialPolitic {
       http://www.semwebtech.org/mondial/10/meta#myCountry
       http://www.semwebtech.org/mondial/10/meta#neighbor
}
```

Note that the concept <mon:myNeighbors> is not listed in the hash. This is due to the fact that no domain node supports this concept. It only exists as a meta class defined within the ontology.

The query broker now collects all the data from the domain node.
In this case this is the information about 'Germany' from the <mon:myCountry> concept:

```
<http://www.semwebtech.org/mondial/10/countries/D/> <http://www.w3.org/1999/02/22-
    rdf-syntax-ns#type> <http://www.semwebtech.org/mondial/10/meta#myCountry>
```

These are some of the neighbor relations from the <mon:neighbor> concept:

```
<http://www.semwebtech.org/mondial/10/countries/D/> <http://www.semwebtech.org/
    mondial/10/meta#neighbor> <http://www.semwebtech.org/mondial/10/countries/DK/>
<http://www.semwebtech.org/mondial/10/countries/D/> <http://www.semwebtech.org/
    mondial/10/meta#neighbor> <http://www.semwebtech.org/mondial/10/countries/PL/>
<http://www.semwebtech.org/mondial/10/countries/N/> <http://www.semwebtech.org/
    mondial/10/meta#neighbor> <http://www.semwebtech.org/mondial/10/countries/SF/>
<http://www.semwebtech.org/mondial/10/countries/R/> <http://www.semwebtech.org/
    mondial/10/meta#neighbor> <http://www.semwebtech.org/mondial/10/countries/BY/>
<http://www.semwebtech.org/mondial/10/countries/LT/> <http://www.semwebtech.org/
    mondial/10/meta#neighbor> <http://www.semwebtech.org/mondial/10/countries/LV/>
```

The main query is now executed on the combination of this data and the Mondial ontology. Since a reasoner is used for this query execution, the correct answer is returned, containing all the neighbors of 'Germany,' the only <mon:myCountry> (in this case given as a list, since the full list in the correct markup is to long to display here):

```
<http://www.semwebtech.org/mondial/10/countries/CH/>
<http://www.semwebtech.org/mondial/10/countries/F/>
<http://www.semwebtech.org/mondial/10/countries/L/>
<http://www.semwebtech.org/mondial/10/countries/PL/>
<http://www.semwebtech.org/mondial/10/countries/CZ/>
<http://www.semwebtech.org/mondial/10/countries/DK/>
<http://www.semwebtech.org/mondial/10/countries/A/>
<http://www.semwebtech.org/mondial/10/countries/B/>
<http://www.semwebtech.org/mondial/10/countries/NL/>
```

5 Implementation

In this chapter the changes to the code of the domain broker and query broker are described, along with the newly implemented interfaces.

5.1 General

The implementation of this thesis has been done in Java. To handle the RDF models and resources the Jena framework has been used to be compatible with the previously existing parts of the MARS framework.
The web pages were written in HTML and JavaScript.

5.2 Overview

Figure 5.1 shows the different components of the domain broker.
The *Domain Broker Servlet* is the interface which handles all request from external nodes.
The actual processing is done by the *Domain Broker Class* or the *Query Broker Class*.
The MARS Frontend is a set of web pages from which the functions of the domain broker can be accessed and in which information about the internal status of the domain broker can be displayed.
Furthermore, it can be seen that the query broker holds so called *QueryBoxes* to handle the data of the queries it processes.

The following sections will provide further details concerning the components.

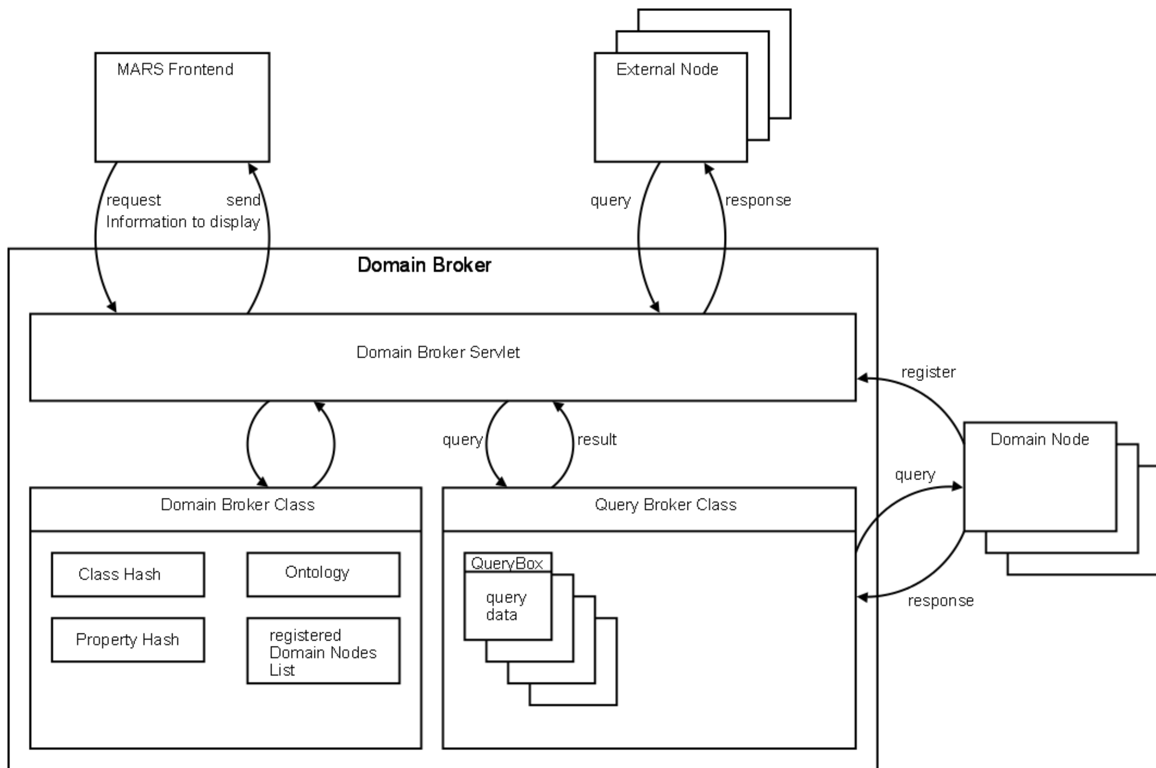


Figure 5.1: The Domain Broker Structure (implementation)

5.3 Domain Broker Servlet

The domain broker servlet provides means to handle the incoming 'GET' and 'POST' request by external nodes. Which method has to be executed is determined by the last part of the URL called by the external node.

The following methods have been added to 'GET' requests:
/query-ids
Returns a list of all stored executed queries to be shown in the MARS Frontend.

/query-parts
Returns a part of a stored query to be displayed in the MARS Frontend. The part returned depends on the 'query-id' and 'query-part' parameter given by the request.

The following methods have been added to 'POST' requests:
/register-domainNode
Registers the domain node of the given URL by calling the corresponding method of the domain broker.
/deregister-domainNode
Deregisters the domain node of the given URL by calling the corresponding method of the domain broker.
/get-hash
Returns the class hash or the property hash, depending on the message given by the request.

The **/register-ontology** method has been removed, since the ontologies are now loaded during the startup of the domain broker.

5.4 Domain Broker Class

Some methods have been added to the domain broker class for the purpose of this thesis. A short overview of the new methods will be given in what follows. All previously existing methods are unchanged.

void loadOntologiesFromLsrToTDB()
This method is called during startup of the domain broker. It acquires the ontology files of the supported domains according to the URL of the domain broker as stated in the LSR. The ontologies are stored in newly created TDB's.

void generateOntologyFromTDB()
After all the ontologies have been stored in the TDBs, this method is called. It builds an N3 string from all the ontologies and sends this to the setOntology() method of the internal ontology class. This is done so that the previously existing ontology class [1] does not have to be changed.
void generateHashes()
This method is called whenever a new ontology is loaded or a domain node is registered. It combines the internal ontology with the support statements of all registered domain nodes and calls the hash factory class to (re-)create both the class hash and property hash.
void registerDomain(String message) and **void deregisterDomain(String message)**
These methods are called whenever a domain node wants to register or dereigister itself. They update the 'ontology and domain node administration structure' (see Section 4.2.4) and handle the TDB's containing the supports statements. After they are done they call the *generateHashes()* method.

5.5 The Hash Classes

Five new classes were created to handle the hashes.

HashFactory
The hash factory contains methods to create and acquire the hashes. All the methods are static, therefore it is not necessary to create an instance of the HashFactory.
ClassHash and ProperyHash
These two classes each contain one hash assigning a HashElement (see below) to a concept. They provide methods to add and remove concepts and get information about them.
ClassHashElement and **ProperyHashElement**
These classes contain the *usesClass*, *usesPrpoerty* and *supportedBy* lists as described in Section 4.3. They provide methods to add, remove and access these lists.

5.6 Query Broker Class

The query broker class handles all parts of the query execution.
During the implementation of this thesis the query execution has been rewritten completely. The previously existing query handling has not been deleted, so that it can be inspected and parts of it may be reused later on.

After receiving a query, the query broker generates a unique number for that query. With this number and the query string, a QueryBox object is created. The content of this object is continuously updated while the query runs through the steps of the query execution as described in Section 4.4.

Each of the mentioned steps is handled in its own method. These methods are straightforward and do exactly what is described in the design section.

The only noteworthy part is the temporally implemented domain node querying: The method **ResultSet queryDomain(String domainNode, String query)** sends the given query to the given domain node and returns the result. Since so far there are not enough real domain nodes for testing, this method fakes domain nodes and actually sends the query to locally stored TDB's. This, of course, needs to be changed as soon as enough real domain nodes are implemented.

5.7 The QueryBox

The QueryBox class was created for the purpose of this thesis. Its purpose is to hold all the relevant data for one query. This includes the following:

- the query,
- the query number ,
- the relevant concepts list,

- the 'supporting domain nodes hash', and
- the final result.

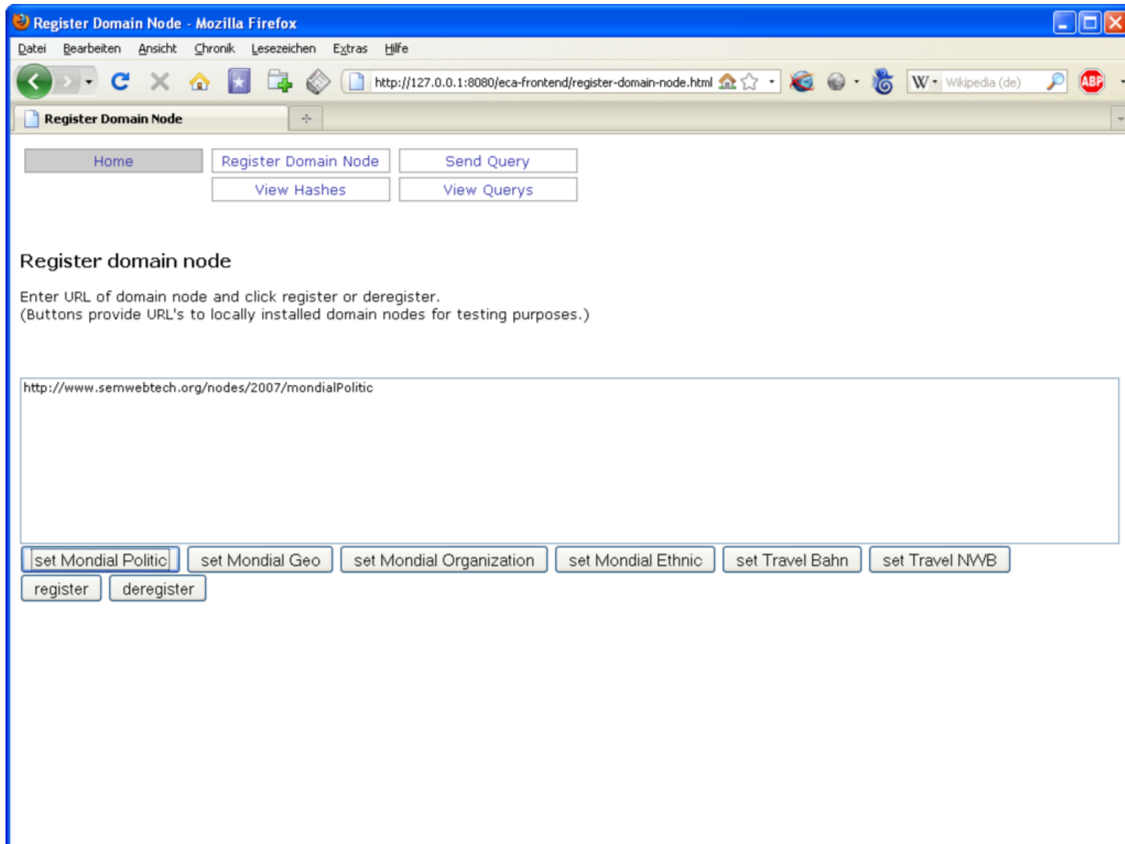
For each query one instance of the QueryBox is created. Due to the fact that the data required to execute a query is stored in an external object, the query broker is capable of executing multiple queries simultaneously with only one instance of the query broker class. If the data were stored in local variables within the query broker this would not be the case.

5.8 MARS Frontend

The MARS frontend is an HTML-based set of web pages to access and test the components of the MARS framework.
The whole communication with the domain broker is done by AJAX and XMLHttpRequest [16] object calls.

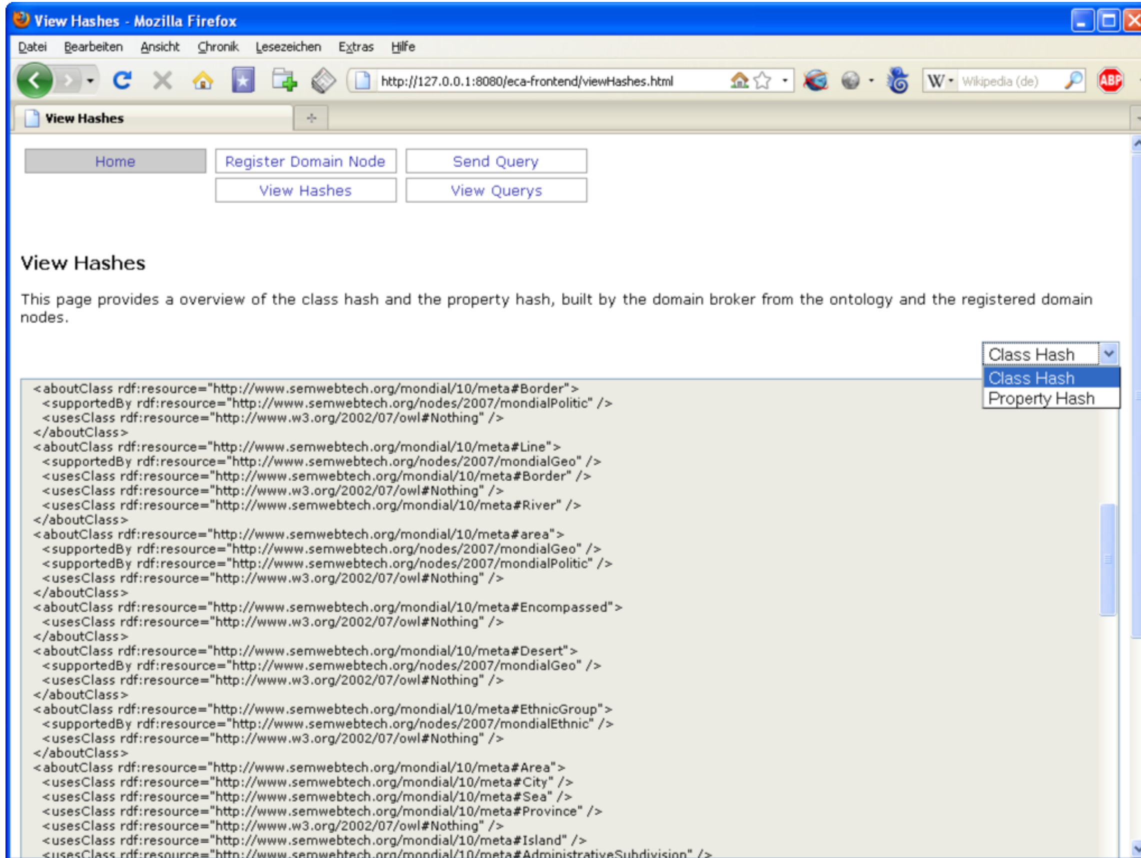
For the purpose of this thesis, a section has been added to the frontend to access the new functions of the domain and query broker. The following pages were added:

Register Domain Node



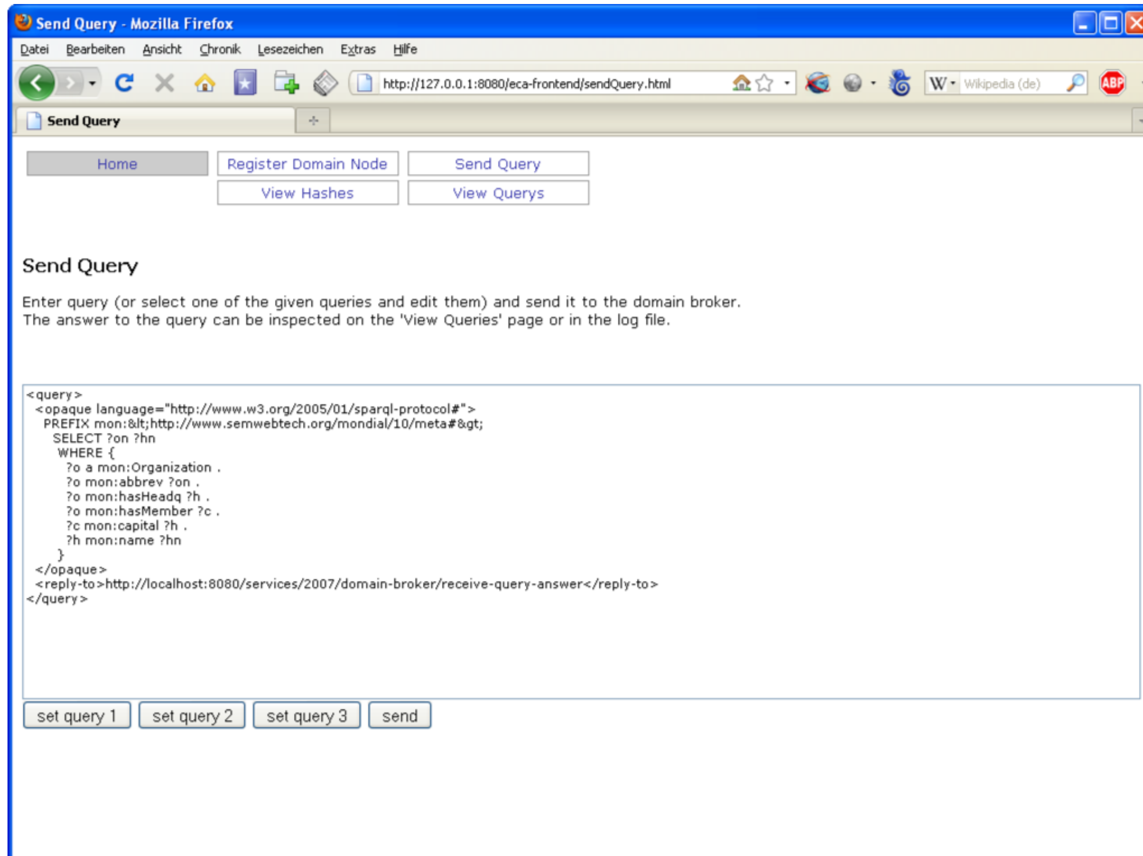
This page provides an interface to register domain nodes. To do so, the URL of the domain node needs to be entered in the text field. If the 'register' button is pressed, the URL will be sent to the domain broker, which will add the node to its registered domain nodes list as described in Section 4.2.3.
A domain node can be deregistered in the same way by pressing the 'deregister' button.

View Hashes



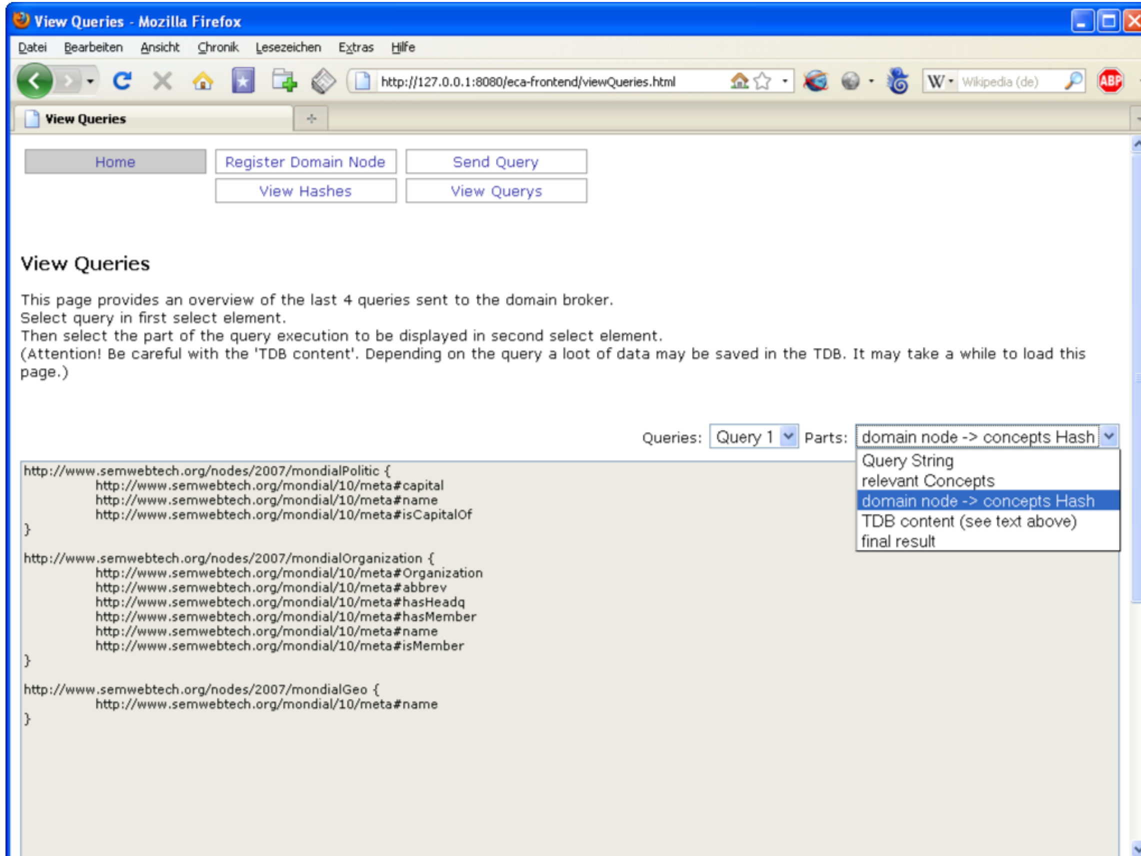
The view hashes page displays the class and property hashes built by the domain broker. The user can check, whether all the concepts in the ontologies were added to the hashes correctly and if the domain nodes were registered with the correct concepts.

Send Query



From this page a query can be sent to the domain broker.
The query can be entered in the text area. The 'send' button sends the query to the domain broker.
The answer to the query can be inspected either on the 'View Queries' page (see below) or in the domain broker log file.

View Queries



This page provides an overview of the last four queries sent to the domain broker. Only four queries are stored because the resource consumption for storing the queries is quite huge. Therefore storing too many queries will considerably slow down the domain broker. The query to be shown can be selected in the first selection box.
The part of the query that should be displayed can be selected in the second selection box. The following parts of the query execution can be displayed:

- **query**: The query string sent to the domain broker.

- **relevant concepts**: The list of relevant concepts, as determined by the query broker.
- **domain node -> concepts hash**: The 'supporting domain nodes hash', created by the query broker.
- **TDB content**: The content of the TDB where the query broker has stored the answers of the subqueries.
- **final result**: The result of the execution of the query send to the domain broker.

6 Conclusion

The design and implementation of the internal logic of a query broker was the main target of this thesis. To accomplish this target the domain and query broker previously designed in [1] were revised and extended.

The handling of ontologies were changed so that their loading procedure now depends on the LSR. Domain nodes within the Semantic Web can now register themselves at the domain broker and therefore enable the domain and query broker to provide a better service to other web nodes.

The query broker can now handle queries to all domains made known to it. The execution of an incoming query is therefore much quicker, due to the use of a hash to expand the concepts within the query and the fact that the subqueries, sent to the data holding domain nodes are created such that they can be executed without the use of a reasoner.

A set of HTML based web pages were created to access the new domain and query broker functionality through HTTP-GET and HTTP-POST requests.

The query broker still needs to query all domain nodes that may contribute data to the execution of the incoming query. This is due to the fact that the query broker has so far no means for determining which domain node may contribute relevant data and which may not. Further work on the query broker may come up with more internal intelligence, so that the query broker does not need to collect these irrelevant data anymore.

The query broker implemented in this thesis supports only the query language SPARQL. This applies to incoming queries as well as to the subqueries sent to the domain nodes.

Further work on the query broker may provide means to support other query languages (for example, querying SQL databases in addition to querying domain nodes).

Bibliography

[1] T. Knabke: Entwicklung und Implementierung eines Domain Brokers für das Semantic Web, Masters Thesis, Göttingen Unversity, 2006
http://www.informatik.uni-goettingen.de/studies/courses/theses.htm?&show=single&thesis_key=117

[2] Naming and Addressing: URIs, URLs, ... : http://www.w3.org/Addressing/

[3] Web Ontology Language (OWL):

[4] Resource Description Framework (RDF): http://www.w3.org/RDF/
http://www.w3.org/TR/owl-features/

[5] RDF Schema (RDFS): http://www.w3.org/TR/rdf-schema/

[6] SPARQL Query Language for RDF: http://www.w3.org/TR/rdf-sparql-query/

[7] TDB: http://jena.hpl.hp.com/wiki/TDB

[8] Apache log4j: http://logging.apache.org/log4j/

[9] Pellet: http://pellet.owldl.com/

[10] HP Labs Semantic Web Research: http://www.hpl.hp.com/semweb/

[11] Wolfgang May, José Júlio Alferes, Ricardo Amador. An Ontology- and Resources-Based Approach to the Evolution and Reactivity in the Semantic Web. In Ontologies, Databases and Semantics (ODBASE), number 3761, pages 1553-1570. Springer, 2005

[12] Reasoning on the Web with Rules and Semantics (REWERSE): http://rewerse.net/

[13] Thomas R. Gruber. *A Translation Approach to Portable Ontology Specifications.* Knowledge Acquisition, 5(2):199-220, 1993.

[14] The MARS LSR fil: http://www.semwebtech.org/mars/2006/lsr.rdf

[15] The Mondial ontology: http://www.dbis.informatik.uni-goettingen.de/Mondial/Mondial-RDF/mondial-meta.rdf

[16] The XMLHttpRequest object: http://www.w3.org/TR/2009/WD-XMLHttpRequest-20090820/