

# Carbonara - a semantically searchable distributed repository

Sima Baymani, [sima@kth.se](mailto:sima@kth.se)  
Emil Stridfeldt, [emst@kth.se](mailto:emst@kth.se)

May 10, 2005

## **Abstract**

This thesis presents Carbonara, a semantically searchable distributed repository. Carbonara is part of the NetSim project at the Swedish Defence Research Agency. The NetSim project aims to enhance information sharing, collaboration and data exchange in a multi organisational environment. Carbonara is a module within NetSim and provides means for data storage and metadata searches to other modules of NetSim.

Through interviews with NetSim project members and demands from the NetSim project, we have created a list of requirements. These requirements were the foundation for the design of the Carbonara system.

A prototype of the Carbonara system has been implemented using the Globus Toolkit and Jena. The metadata languages used are RDF and OWL. The Globus Toolkit is a middleware for grid networks used to provide the Carbonara system with low level operations for data storage and lookup. Jena is a framework for storing and handling metadata used to create the semantic database over which metadata reasoning and querying is performed. RDF and OWL are languages for creating information models and resource metadata.

Grid systems are a good middleware for designing applications in a multi organisational environment. They provide security and data management tools suitable for virtual organisations. Semantic searches are highly efficient because of the possibilities to find data through associations rather than keywords. By adding reasoners to a semantic database, hidden relations can be found.

Although the functionality of the technologies used are desirable, the frameworks implementing these technologies are not yet performing well enough. However, we encourage further research and development in this area.

# Acknowledgements

We would like to thank:

Our examiner Rassul Ayani, IMIT/FOI, for pointing this project out, input and proofreading of this Master's thesis.

Our supervisor Marianela García Lozano, FOI, for giving us this project, mentoring the design, for input and proofreading of this Master's thesis and finally for being professional at all times.

Martin Eklöf, Martin Gülich and Pernilla Svan, FOI, for help and discussion in the semantic area.

Farshad Moradi and the rest of the NetSim group, FOI, for help with the requirements on the Carbonara system.

Gabriel Ghinita, NUS, for help with all matters concerning the Globus Toolkit.

Thomas Sandholm and Olle Mulmo, PDC, for encouraging us when in doubt about the Globus Toolkit.

Niklas Wallin, FOI, for making the network lab available to us.

All personnel at the Department of Systems Modelling, FOI, for always having their doors open to us.

Jan Aston, FOI, for network support.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	NetSim . . . . .	1
1.2	Problem Statement . . . . .	2
1.3	Method . . . . .	2
1.4	Overview of Report . . . . .	3
<b>2</b>	<b>Distributed Repositories</b>	<b>5</b>
2.1	Peer-to-peer Systems . . . . .	5
2.1.1	Pastry . . . . .	6
2.1.2	PAST . . . . .	7
2.1.3	Gnutella . . . . .	7
2.2	Server-based Systems . . . . .	8
2.2.1	Andrew File System . . . . .	9
2.3	Grid Systems . . . . .	9
2.3.1	Open Grid Services Architecture . . . . .	10
2.3.2	Open Grid Services Infrastructure . . . . .	10
<b>3</b>	<b>Metadata and Related Technologies</b>	<b>12</b>
3.1	Metadata Related Terms . . . . .	12
3.1.1	Metadata . . . . .	13
3.1.2	Information Models . . . . .	13
3.1.3	Semantic Searches . . . . .	14
3.1.4	Inferencing . . . . .	14

3.2	Metadata Languages . . . . .	15
3.2.1	The Resource Description Framework . . . . .	15
3.2.2	The Resource Description Framework Schema . . . . .	16
3.2.3	The Web Ontology Language . . . . .	17
3.3	Metadata Stores . . . . .	17
3.3.1	Jena . . . . .	17
3.3.2	Sesame . . . . .	18
<b>4</b>	<b>Requirements and General Design</b>	<b>20</b>
4.1	Interviews . . . . .	20
4.2	Requirements . . . . .	21
4.3	General Design . . . . .	22
4.4	Choice of Platform . . . . .	23
<b>5</b>	<b>Globus Toolkit 3</b>	<b>25</b>
5.1	Grid Security Infrastructure . . . . .	25
5.1.1	Authentication . . . . .	25
5.1.2	Identity Converting . . . . .	26
5.1.3	Delegation and Single Sign-On . . . . .	26
5.1.4	Confidential Communication . . . . .	27
5.2	GridFTP . . . . .	27
5.3	Replica Location Service . . . . .	29
5.3.1	Replica Location Service Framework . . . . .	29
5.3.2	Common Replica Location Service Server . . . . .	30
5.3.3	Soft State Updates . . . . .	31
5.4	Globus Toolkit and Grid Services . . . . .	31
5.4.1	Web Services . . . . .	31
5.4.2	Grid Services . . . . .	32
5.4.3	Globus Toolkit with Grid Services . . . . .	32

<b>6</b>	<b>The Semantic Tools</b>	<b>33</b>
6.1	RDF and OWL . . . . .	33
6.1.1	RDF . . . . .	33
6.1.2	OWL . . . . .	35
6.2	Reasoning . . . . .	38
6.3	Jena . . . . .	39
6.4	RDQL . . . . .	40
<b>7</b>	<b>Design</b>	<b>42</b>
7.1	Architecture . . . . .	42
7.1.1	Resources . . . . .	42
7.1.2	A Carbonara Node . . . . .	43
7.1.3	The Carbonara Network . . . . .	44
7.2	Resource Metadata . . . . .	46
7.2.1	Information Models . . . . .	47
7.2.2	Inferencing . . . . .	48
7.2.3	Semantic Database . . . . .	48
7.2.4	Administrative Information . . . . .	48
7.2.5	Descriptive Information . . . . .	50
7.2.6	User Management . . . . .	50
7.2.7	Access Control . . . . .	51
7.3	Version Handling . . . . .	52
7.3.1	Versions . . . . .	52
7.3.2	Branches . . . . .	53
7.4	Characteristics of Operations . . . . .	54
7.4.1	Resource creation . . . . .	54
7.4.2	Resource removal . . . . .	54
7.4.3	Resource update . . . . .	55
7.4.4	Searching . . . . .	55

<b>8</b>	<b>Implementation</b>	<b>56</b>
8.1	Environment . . . . .	56
8.2	Prototype Design . . . . .	57
8.3	Service Modules . . . . .	59
8.4	CarbonaraCore . . . . .	59
8.4.1	Find Resource . . . . .	59
8.4.2	Lookup Resource Location . . . . .	60
8.4.3	Download Resource . . . . .	60
8.4.4	Create Resource . . . . .	60
8.4.5	Set Resource Description . . . . .	61
8.4.6	Get Resource Description . . . . .	61
8.4.7	Set Resource Data . . . . .	61
8.4.8	Remove Resource . . . . .	62
8.5	Carbonara Grid Service . . . . .	62
8.6	Performance Tests . . . . .	62
8.6.1	Testbed . . . . .	63
8.6.2	Creation and Removal . . . . .	64
8.6.3	Search . . . . .	67
8.6.4	Comments . . . . .	68
<b>9</b>	<b>Conclusions and Future Work</b>	<b>69</b>
9.1	Main Achievements . . . . .	69
9.2	Evaluation . . . . .	69
9.3	Comments on the Chosen Platform . . . . .	71
9.4	Future Work . . . . .	71
9.5	Conclusions . . . . .	73
	<b>Appendices</b>	<b>75</b>
<b>A</b>	<b>Requirements List</b>	<b>75</b>
A.1	Underlying Platform . . . . .	75

A.2	Architecture . . . . .	76
A.3	Functionality . . . . .	77
A.4	Usability . . . . .	77
A.5	Security . . . . .	78
<b>B</b>	<b>Flow diagrams</b>	<b>79</b>
B.1	Set resource description flow diagram . . . . .	79
B.2	Set resource data flow diagram . . . . .	80
B.3	Create resource flow diagram . . . . .	81
<b>C</b>	<b>Environment</b>	<b>82</b>
<b>D</b>	<b>Glossary</b>	<b>83</b>

# Chapter 1

## Introduction

This thesis describes a Master's project conducted at the Swedish Defence Research Agency. It presents Carbonara, a searchable distributed repository. Carbonara is one of the components that are part of the current implementation of the NetSim platform[14].

### 1.1 NetSim

NetSim stands for Network Based Modelling and Simulation, and is a project at the Swedish Defence Research Agency. It aims to develop a platform for information sharing, distributed collaboration and reuse of data, in the area of Modelling and Simulation.

NetSim supports reuse of code and facilitates advanced distributed collaboration. Users can communicate over long geographical distances, but still have the opportunity to use advanced tools to build up strategies, projects or simulations. Simulations that would take days or even weeks to execute could be run in a distributed mode by a group of computers, speeding up the total execution time. Components, code, simulations and other data can be shared, increasing the reuse of data and collaboration between groups.

The vision of NetSim is to be a platform useful for a broad number of areas, spanning from distributed collaboration to federated simulations. The project demands modular implementation of its different tools, to simplify further development and enable exchange of components.

## 1.2 Problem Statement

One component of NetSim is a searchable distributed repository, yet to be implemented. The purpose of the repository is to be the storage module of NetSim. The repository is a hybrid of a filesystem and a library, to ease resource storage and retrieval in NetSim. A basic function of this repository, apart from up- and download, is to store and retrieve information about resources of any kind. The retrieval is founded on a semantic search, where users query the kind of resource they are interested in by stating properties they are required to have and their relationships with other resources. This means that in order to enable semantic querying, resources must be described by metadata.

This reveals functionality similar to a database, with added interfaces for uploading and downloading data, setting access rights etc. Other aspects of consideration are robustness, consistency and versioning. The goal of the thesis project is to find a working platform and upon it build the functionality we need. Thus the assignment is not to build the system from scratch, but to use existing applications and integrate them. The repository is a module that will be fit in with other tools that are part of the NetSim project. This means that it must have a generic design, to enable the other tools to use the capacities of the repository for their own needs.

## 1.3 Method

The Master's project was divided into several phases. In the first phase we identified the requirements of the system we were about to build. To do this, we interviewed NetSim project members and studied literature on platforms that could meet our needs. Based on these interviews and platform studies, we formed a requirements list, together with priorities for each requirement. A general design was outlined and a final choice was made about which platform to use.

In phase two we created the specific architecture of our project based on the general design and the chosen platform. A major part of this phase was to set up and learn how to use the tools chosen for the architecture.

The third phase was about implementing a prototype for the architecture outlined in phase two. The goal of the implementation was to integrate the different subsystems and on top of this add our specific file system.

In the fourth phase we tested our prototype for functionality and performance. We checked that all operations worked correctly and had the expected result. We measured the time needed to perform operations for creation, removal and querying of resources.

The report has been written in bursts between and during these phases.

During installation and implementation, the work was distributed between us. Emil has worked with the parts concerning file transfer, file lookup and storage functionality. This included working with the Globus Toolkit and grid services. Sima has worked with the semantic parts such as the semantic database, reasoners and the information model. This involved using Jena and OWL. Everything else has been carried out by both of us.

## 1.4 Overview of Report

The rest of this report is structured as follows:

**Chapter 2** Introduction to distributed storage systems. We discuss peer-to-peer, server based and grid systems and explain how they work. We describe implementations for each system type. For each system advantages and disadvantages are discussed.

**Chapter 3** We introduce terms used in the semantic area together with metadata languages. We also describe applications managing semantic information. We explain what reasoning is and how it works.

**Chapter 4** In this chapter the requirements on the Carbonara system are explained. We describe the general design of the Carbonara system and which applications we chose to build it on.

**Chapter 5** Introduction to the Globus Toolkit. The tools from Globus that are used in this project are described. These include the Grid Security Infrastructure, the GridFTP and the Replica Location Service and are thoroughly explained. An introduction to grid services is also given in this section.

**Chapter 6** A more thorough explanation of the metadata languages RDF and OWL. Deeper description of Jena, reasoning and Jena's query language RDQL.

**Chapter 7** This chapter states the final design of the Carbonara system. We describe the architecture of the system and the resources that populate it. We also explain the functionality of the system, i.e. the operations available.

**Chapter 8** The prototype implementation and tests are described here. We discuss which parts of the design in Chapter 7 that have been implemented. We describe the modules which provide the operations and how the operations are carried out. The tests are thoroughly explained and the results are discussed.

**Chapter 9** We evaluate the system design and the platform applications. Suggestions for future work are given together with an overall comment on the project.

**Appendix A** A deeper description of the requirements on this project and the Carbonara system.

**Appendix B** Flow diagrams over the Carbonara operations set resource description, set resource data and create resource.

**Appendix C** A detailed list of the applications and systems that were used by this project.

**Appendix D** A glossary for the acronyms used in this report.

## Chapter 2

# Distributed Repositories

There are many different kinds of distributed repositories, all based on different architectures and semantics. We have chosen to only mention approaches that are widely known and for which we have found open source releases. During our study we came across a great number of systems, architectures and projects, most of them mentioned at the references appendix. Some projects were not open source, some were not implemented and yet some were no longer ongoing. We have, however, not looked too deep in commercial, non open source projects since a requirement on the project was that all tools used must be open source. This means there may exist non open source systems better suited for us. We have also abandoned systems that clearly do not have the vision of becoming (or being) platform independent. This chapter briefly goes through the systems we listed as most probable to use in Carbonara.

### 2.1 Peer-to-peer Systems

A peer-to-peer (P2P) network is a communication model where each participating node has the same capabilities and each party may initiate a connection. This model differs from the client-server model or master-slave model where participants have different responsibilities. In the peer-to-peer model each peer acts both as a server and a client.

Peer-to-peer systems can be divided into unstructured and structured systems.

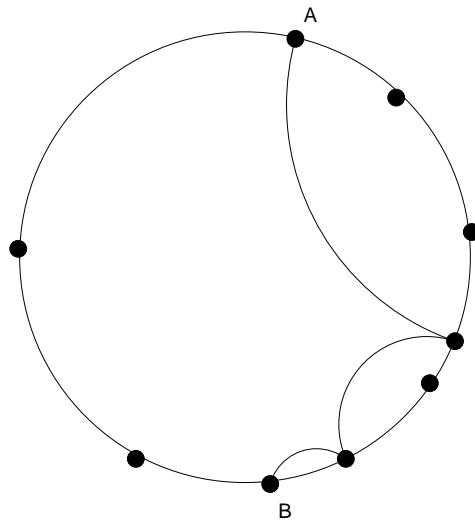
In an unstructured peer-to-peer network, nodes randomly connect to other nodes in the network. Each node maintains an ad-hoc list of other nodes. This list is the only knowledge the node has about the network. Generally, message passing is constructed in a flooding manner. Each node redirects messages to all other nodes in its ad-hoc list. Therefore no guarantee is made for delivery of a message.

Nodes in a structured peer-to-peer network join using a specific protocol. Leaving the network is system specific. A node manages some kind of routing information for reaching other nodes in the network. The exact design of the routing information is system specific. This routing information reduces the logical network overhead when sending a message. A message is normally sent only to one other node. That node is chosen by using the routing information and system specific routing algorithms.

### 2.1.1 Pastry

A decentralised self stabilising overlay network called Pastry [34] is one of many research projects in peer-to-peer computing. Pastry is organised as a concept of a distributed hash table. It can be used as an underlying infrastructure for distributed applications. A distributed hash table offers a mapping between keys and values.

Messages are passed from node to node reducing the distance from the present node to the destination. This is done by ensuring that each new node has more information about the destination than the previous node. The routing table in Pastry is constructed to give a number of hops which is logarithmic in the number of nodes available in the system. A simple routing scenario is found in Figure 2.1.



**Figure 2.1:** Sending a message from node A to node B using Pastry routing algorithm

The identifier space is ordered as a modulo ring with size  $2^{128}$ . This means that 0 comes after  $2^{128} - 1$ . Computers which participate in the system are assigned an identifier between 0 and  $2^{128} - 1$  which places them somewhere on the ring. If a hash function is used to assign identifiers, nodes will be spaced evenly throughout this ring.

The identifier space is divided into partitions. The identifier space between two nodes is divided into two equal halves. Each node is responsible for the closest half. This means that an identifier belongs to the numerically closest node.

Each piece of data inserted into the system is assigned an identifier, and is saved on the node responsible for that identifier. The identifier can for instance be computed by using a hashing function on the data.

### 2.1.2 PAST

PAST [13] is a storage utility built on top of Pastry. A hash function calculates the unique identifier for each file stored in PAST. The file is then stored at the node corresponding to the unique identifier located by using the Pastry routing scheme. PAST does not supply any catalogue structure or additional file information. The unique identifier is the only key used for retrieving a file stored in PAST.

PAST uses a replication scheme in order to guarantee no loss of data when a node is leaving the system. The file is stored at the  $k$  numerically closest nodes with respect to the unique identifier of the file. The size of  $k$  depends on the importance of the file stored in PAST. This replication policy in addition to the routing scheme reduces the number of hops when performing lookup on a file. Because of the logarithmic behaviour of the routing scheme it is likely that one of the numerically closest nodes will receive the lookup request before the destination node. If the node then has a copy of the file it is returned and the request is not sent forward.

The semantics for removing a file is somewhat weaker than a traditional storage system. Only the owner of a file is entitled to remove it. After removal of a file, the file can not be guaranteed to be retrieved again. It is not guaranteed that the file is removed on all nodes storing it. This weaker semantic is used because no agreement protocol on removal is used by the storing nodes.

The security in PAST depends on public/private key pairs. The creation of the unique identifier for a file depends upon the public key of the creator. Files can be encrypted before storage on nodes to reduce risk of unwanted intervention.

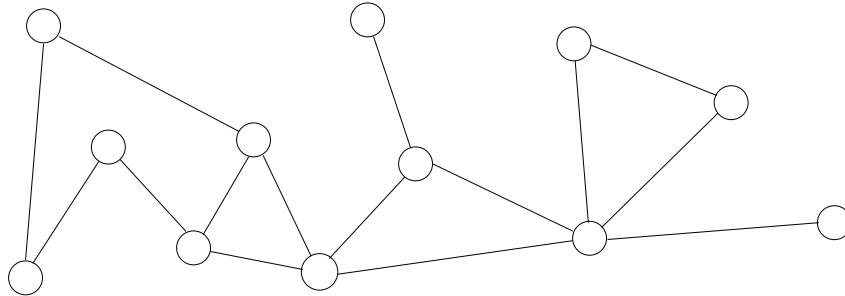
### 2.1.3 Gnutella

Gnutella [23] is a protocol for unstructured peer-to-peer overlay networks. The original protocol was redrawn and the one in use is a backward-engineering of the original protocol. Several applications have implemented this protocol such as LimeWire [24], BearShare [3] and others.

Gnutella is a decentralised protocol. A node has the same responsibilities in the system as any other node. Each node maintains a node list. This list is used for

sending messages in the network. When a node receives a message it forwards it to all nodes in its node list except the sending node. If the node has a reply to the message it sends a reply to the original sender. Each message has a time to live parameter which is decreased at every node visited.

A node joins the network by contacting a known node in the network. It receives a node list from the known node. The joining node then contacts nodes on this list until a predefined threshold is reached. The remaining nodes on the list are kept in case of contacted nodes leaving the network. A sample Gnutella network is found in Figure 2.2.



**Figure 2.2:** A sample Gnutella network

The lack of structure of the network in combination with the time to live parameter on a message can not guarantee that a message is delivered to a specific node. A node does not reply if it did not receive a message either. This means that searching for information in a Gnutella network can not state that the information does not exist in the network. The searching node does not know if the queried node does not have the information, has the information but is temporary unavailable or if it has the information but the message did not reach it. For a file sharing service this behaviour is clearly acceptable since it is in use but for a file system it is not.

## 2.2 Server-based Systems

A server-based distributed repository has the usual client-server approach, where computers configured as servers offer specific services to computers configured as clients. These roles are not to be changed, i.e. clients cannot act as servers, as in the case of peer-to-peer systems. The servers share the workload, distributing data and requests between themselves. However, none of this distribution is apparent to the client, which sees the servers as one single unit.

### 2.2.1 Andrew File System

The Andrew File System (AFS) [28], is built up by a cluster of file servers called a cell. Typically an organisational unit has its own cell. AFS organises cells under a single (global) name space. A user can access this name space by opening the local directory `/afs`. Under `/afs` different cells are mounted, each under its own directory, revealing a portion of the global name space belonging to a certain organisational unit. In this way clients can be configured to mount cells on different networks, belonging to other organisations, but still giving the user the illusion of a single file system. OpenAFS [29] and Coda [35] are based on AFS and are freely available.

AFS gives organisations the possibility to share file space in a way that is both secure and user friendly. Implementations are fully developed, and versions exist for at least \*nix and Windows systems. The Coda implementation also has extended replication of data, making it even more robust to use over unreliable networks. The fact that there are implementations that are open source and freely available is also an advantage.

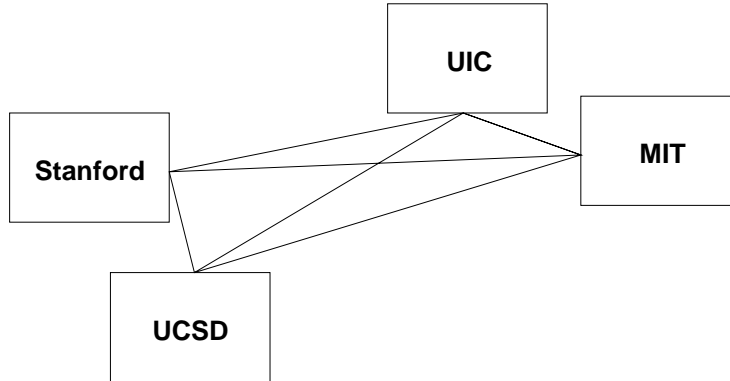
AFS systems need administration. To mount new cells, clients must be configured to show the new cells. Users also have to be authorised to access the new cell. Therefore the coming and leaving of cells will be expensive on all dimensions. NetSim clients will come and go at will, and the file system must comply and cope with that. There can not be too much administration, so that single users cannot deal with it themselves, or that an organisation must supply the system with an administrator.

## 2.3 Grid Systems

A Grid is a “*distributed computing infrastructure that supports the creation and operation of virtual organisations by providing mechanisms for controlled cross-organisational resource sharing*” [15].

Grid systems are a technology mostly used by the science community for sharing large data sets and to perform large scale simulations and calculations. A typical grid network consists of several computer centres, i.e. universities that are connected through a high bandwidth connection. A typical grid network is shown in Figure 2.3.

The concept of virtual organisations is used frequently in grid technology. Organisations need to share resources among each other. These resources are not generally files. Instead it might be direct access to computers, software, data or specific instruments. Users with access rights and service providers are clearly defined with respect to security. A set of users and service providers makes a virtual organisation. A virtual organisation makes it possible for a disparate group of users to share resources in a secure and controlled fashion in order to



**Figure 2.3:** A typical grid network

achieve a shared goal.

There are several important features in grid computing worth mentioning. The ability to sign in just once in a grid system is important. A grid user would normally use several grid resources. Instead of signing in for each resource or security domain, the first sign in is used in all the resource invocations. When invoking a resource, this resource might need to access other resources. The resource must then act on behalf of the user. A user can use a chain of resources by simply contacting the first one. The security model of the grid must then interoperate with different heterogeneous local security systems.

### 2.3.1 Open Grid Services Architecture

The Open Grid Services Architecture (OGSA) [16] defines standard procedures for creating, naming and discovering of transparent grid services. It provides local transparency, multiple protocol bindings for service instances and supports integration with underlying native platform facilities. OGSA combines technologies and concepts from both grid and web services communities. The OGSA standard is handled by the Global Grid Forum[18].

### 2.3.2 Open Grid Services Infrastructure

The Open Grid Services Infrastructure (OGSI) [37] gives a formal and technical specification of what a grid service is. OGSI specifies all grid service interfaces defined in OGSA. These interfaces are specified using Web Services Description Language (WSDL). This language is also used by web services to specify interfaces. The OGSI standard is also handled by the Global Grid Forum. The Globus Toolkit [17] provides a reference implementation of the OGSI. This im-

plementation is provided in the Globus Toolkit along with other non OGSF compliant services [17].

## Chapter 3

# Metadata and Related Technologies

One of the demands of the NetSim project was that all objects stored in the repository should be semantically searchable. Our directions were that each object would have some sort of metadata description bound to it, and that the semantic searches would be based on this metadata. Questions that arise in the light of these requirements are “What is a semantic search?” and “What is metadata?”. We will try to answer these questions here together with an overview of technologies we found interesting to use.

### 3.1 Metadata Related Terms

This section will introduce the terms used to describe the semantic part of the Carbonara system. An important note to make is that because of the nature of the concepts that are defined here, the definitions in this report may differ from definitions found elsewhere. The concepts are used in several different areas (philosophy, biology, etc), thus it is difficult to give a definition or meaning that is applicable to all areas. We also give our definitions from a practical perspective: how we use the concepts to design and implement our system. To give the reader the possibility to form an own opinion of the matters, each section is ended with references to alternative definitions. A good list of definitions on the web for most of the terms can be produced by using the “define:”-mechanism of *Google*<sup>1</sup>.

---

<sup>1</sup>[www.google.com](http://www.google.com)

### 3.1.1 Metadata

Everything in the world around us can be described in terms of different attributes and relations. As humans, we have learnt how to express and process descriptions of objects. We have learnt to describe the colour, shape and sound of cars, the taste and consistency of food and the motif of an image. We can classify objects and group objects together according to their relations or similarities. However, these characteristics are mostly only visible to humans, and the descriptions in turn are also mostly only comprehensible to humans. We would like for a computer program to understand what an image depicts, if it is in black/white or colour, who made the image, and other properties and relations that the image has. This is where *metadata* plays an important role.

Metadata is data about data. In our context, metadata describes the characteristics and relations of an object in a formatted way. Metadata are terms that are associated with a meaning, and may be used in the object description process. By using metadata, object attributes and relations are made machine understandable, enabling automation of description processing. Metadata can be *structured* in several ways: in hierarchies, as simple sets or any other way useful for the situation. There are also several forms of representation, languages for metadata. What is important is that the structure and the chosen form of representation must be expressive enough to describe the objects and their relations in an accurate way.

Alternative explanations and definitions can be found in *Practical RDF* [31, Ch 5] and *Wikipedia* [42].

### 3.1.2 Information Models

In the previous section we defined metadata to be data about data and we also mentioned that metadata can be structured. By structure we mean a predefined ordering and grouping of metadata terms, that determines how to use those terms. A class hierarchy with properties for each class is one sort of metadata structure. In this report we will define an *information model* to be a metadata structure associated with *rules* on how to interpret the structure. The rules can define constraints, build up hierarchies or create sets of related objects. We use *ontology* as a synonym for information model.

Objects that are described using terms from a specific information model are said to be *instances* of that model. A set of information models together with instances of those models form a *knowledge base*. We can compare the knowledge base to a relational database: the information models of the knowledge base is analogous to the database *schema*, the instances of the knowledge base are analogous to the *tuples* of the database. The schema defines how the information in the database should be structured in the same way as the information model defines how metadata should be structured. Because of this, the word *schema* is often used as a synonym for information model, when implementing a knowledge

base. We use the term *semantic database* as a synonym for knowledge base.

Alternative definitions of the terms mentioned in this section can be found in *Wikipedia* [43] [41], the *OWL Guide* [27], the *OWL Use Cases and Requirements* document [20], *Practical RDF* [31, Ch 12] and a paper by T. Gruber [19].

### 3.1.3 Semantic Searches

The searching of information is often based on *keywords*, literals that must have a match. The weakness of keyword based searches are that there may be several keywords of interest, maybe too many, or the keywords have several meanings depending on domain. A *semantic search* concentrates mainly on *concepts* instead of keywords. When a keyword based search asks for a car with a specific model name, the semantic search asks for cars of a specific *type*. By using concepts, i.e. types or classes, we broaden the search without diluting the result with irrelevant answers. A keyword based search can often be too narrow or too wide since it is difficult to balance the right keywords. A semantic search can easier be balanced because it is concept based. It is possible to narrow or broaden the search by using a more specific or more general concept.

A semantic search is performed on a knowledge base. The concepts used in a semantic search are terms from the information models used in the knowledge base. The terms in the search have thus been given a well defined meaning and their relations to other terms have been specified. The results of the search can be instances of and terms from the information models. If we have a class hierarchy for cars and engines, examples of searches could be “Which types of cars are there?” or “Which cars of type A have an engine of type B?”.

### 3.1.4 Inferencing

We define the process of *inferencing* as deducing new information from already known information by following predefined rules. An *inference engine* operates on a knowledge base to derive additional information about the instances in the knowledge base. The deduction of the new information is based on the information model of the knowledge base, and the rules defined for that information model. These rules may state what can be inferred if some precondition is satisfied, and if any other rules can be triggered based on this inference. Another word for inference engine is *reasoner*, which yields the verb *reasoning* as a synonym for inferencing.

Assume that we have defined an information model with some classes and properties. Examples of rules may then be:

**Rule 1** *If  $x$  is an instance of Class A, and Class A is subclassOf B, then  $x$  is an instance of Class B also.*

**Rule 2** *If property  $P$  is symmetric and an instance  $x$  is related to an instance  $y$  by property  $P$ , then  $y$  is related to  $x$  by property  $P$ .*

By applying these rules, new statements can be made about the given knowledge base, completing it further. Information that was not given or not visible when constructing the knowledge base can now be deduced and added. This gives us the possibility to see what we have overseen, things that can not be and things that we did not think of. Inconsistencies can be discovered and corrected. Querying a knowledge base that is connected to a reasoner can yield answers that would not be visible without the reasoner, since the reasoner adds inferred statements.

Alternative explanations of inferencing and inference engines can be found in *Wikipedia*[39][40].

## 3.2 Metadata Languages

As mentioned in the beginning of this chapter, objects stored in the Carbonara system need a description. A suggestion from the NetSim project was that the Resource Description Framework (RDF) is used as metadata framework, since it is well known and widely used. It is also the standard framework for web based metadata, stated by the *World Wide Web Consortium* [38]. RDF is used to express metadata, but it does not provide the means to define new metadata terms, i.e. design an information model. Two languages that extend RDF to build up information models are the Resource Description Framework Schema and the Web Ontology Language, both described below.

### 3.2.1 The Resource Description Framework

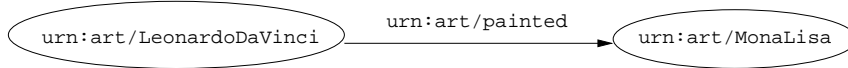
The Resource Description Framework (RDF) is a structure for expressing metadata. Primarily RDF is used “for representing information about resources in the World Wide Web”[25], but can be used to address a wide area of problems, not only web related. The RDF syntax is based on *triples*, called *statements*. Statements describe something about a *resource*. A resource may be anything of interest for the application to describe: a web page, a car, an image. The statement describes an attribute of the resource. Each statement contains a subject, predicate and an object, similar to a sentence in a spoken language. The subject is the resource the statement is about, the predicate is a property that the subject has or is, and finally the object is the value of the property. Objects can be either resources or literals. Below is an example:

*Leonardo da Vinci painted Mona Lisa.*

This is a statement, with “*Leonardo da Vinci*” as subject, “*Painted*” as predicate and “*Mona Lisa*” as object. We are assuming that both “*Leonardo da Vinci*” and “*Mona Lisa*” are resources. Resources and properties are identified by Uniform Resource Identifiers (URIs). In the example below we have used Uniform Resource Names (URNs). They are a subset of URIs used for naming. Using URNs the above resources and properties would perhaps become:

```
urn:art/LeonardoDaVinci
urn:art/painted
urn:art/MonaLisa
```

The official modelling technique of RDF statements is graphs, where subject and object are nodes in the graph, and the predicate is a directed arc connecting them. Resources are depicted as ellipses and literals as rectangles [25, Ch 2.2]. In Figure 3.1 we can see the graph of two resources connected by a property. Triples can also be connected into a bigger graph, by using the fact that the object of one triple can be the subject in another. A RDF knowledge base consists of information models and instances expressed with RDF statements. Such a knowledge base can thus be expressed as a RDF graph, comprising of subgraphs for the information models and the instances.



**Figure 3.1:** RDF graph of the statement “*Leonardo da Vinci painted Mona Lisa*”.

There are several serialisation techniques for RDF, among others RDF/XML [4], Notation3 [6] and N-triples[5]. The official serialisation technique is RDF/XML, but Notation3 and N-triples are more human readable.

### 3.2.2 The Resource Description Framework Schema

The RDF Schema (RDFS) [7] is an extension of RDF. It provides the means for building an application specific information model. RDFS enables the definition of classes and properties and allows them to be ordered in a hierarchy of inheritance. The information model can then be used to describe resources, which become instances of that information model.

The intention of RDFS is to add inferencing capabilities to metadata expressed with RDF, since RDF itself does not provide it. The class and property hierarchy allows for inference of resource types and resource attributes. Properties can also be defined to have a domain and a range, which controls their use and provides additional inferable information. The only constraints RDFS can put on classes and properties are through inheritance and domain-range control. Thus, it can happen that a resource is of type Car and Person at the same time, although this was not the intention of the information model designer.

### 3.2.3 The Web Ontology Language

The Web Ontology Language[26], abbreviated as OWL<sup>2</sup>, extends RDFS with new capabilities for designing information models. The features of OWL enable a much more fine grained definition of information models and how the terms of the model are related to each other. When RDFS only states a hierarchy of classes and properties, OWL defines among others relations between classes, cardinality restrictions and characteristics of properties. Thus it is possible to control the information model and how it is used in the resource description process. As an example, it is possible to state that the classes Car and Person are disjoint, which implies that a resource can never be a Car and a Person at the same time.

OWL is divided into three subsets, each with increasing degree of complexity: OWL Lite, OWL DL<sup>3</sup> and OWL Full. Only a subset of the OWL language is available to OWL Lite, and both OWL Lite and OWL DL have some restrictions on the use of the features of the language. OWL Full has complete access to all features of the OWL language, without any restrictions.

Because of the possibilities of expression with OWL, the language is also complex. Inferencing over any of the sublanguages of OWL is not easy, e.g. the problem of inferencing over OWL Full can not be solved in polynomial time. However, there are functional reasoners for both OWL Lite and OWL DL.

## 3.3 Metadata Stores

The descriptions of the objects in the Carbonara system need to be stored somehow. For this reason we have investigated RDF based metadata storage applications. Beside the storage function, important aspects are which metadata languages that are supported and the possibilities for inferencing and querying over the stored data. It is also important how the *Application Programmer Interface* (API) is built up and how expressive it is. Another important aspect is the possibility of distribution of the stored metadata.

### 3.3.1 Jena

Jena[21] is an open source framework for handling and storing metadata models. Jena is written in Java and provides means for programmatically modifying an information model and its instances. The languages supported are among others RDF, RDFS and OWL.

Jena comes with several internal reasoners. They span from minimal reason-

---

<sup>2</sup>The acronym is actually WOL, but the creators found OWL to be better.

<sup>3</sup>DL stands for Description Logics.

ers only offering domain/range inference, through the Generic Rule reasoner for which rules can be defined by the user, to an OWL reasoner offering inferencing for OWL Lite. It is also possible to connect an external reasoner to Jena through the standardised *DIG interface*[12] developed under the DL Implementation Group[11]. This enables connections to reasoners such as *Racer*[32] which can handle more complex languages such as OWL DL.

Jena provides the possibility to either process knowledge bases in main memory or to persist them using a relational database back end. Depending on the database management system (DBMS) used, it is possible to distribute stored metadata. Jena itself is not distributed but by using a distributed database back end, the whole system may be distributed. It is also possible to write entire knowledge bases to a file. Files containing information models and instances can be loaded and manipulated.

Jena implements its own query language for RDF called *RDF Data Query Language* (RDQL)[33], which enables semantic and keyword based searches on the available knowledge bases.

The API of Jena is large and offers many possibilities. Since Jena supports several languages, there are interfaces for increasing levels of complexity: from simple RDF graphs to complex OWL ontologies. Knowledge bases can be built up and modified programmatically.

### 3.3.2 Sesame

Sesame[8] is an open source RDF database with support for RDF Schema inferencing and querying. It is implemented in Java and has a layered architecture, which makes it flexible and extendable. The metadata languages supported are RDF and RDFS. Sesame provides a full RDFS inference engine and a custom inferencer for which rules can be externally defined. The Sesame development team has plans on providing DIG interface support and OWL support in the future releases of Sesame. Meanwhile it is possible to use extension packages, e.g. for partial OWL support, provided by third party.

Sesame can be used on top of several different storage systems, such as databases and file systems, but it is also possible to process metadata in memory. There have been attempts to distribute Sesame, and research is done within this area[36]. The distribution of data stored through Sesame is also depending on the database system used: a distributed database system results in the distribution of Sesame.

There are several query languages available for Sesame: RQL [22], RDQL [33] and SeRQL [9]. RQL was originally developed by ICS-FORTH<sup>4</sup> in Greece and is partially supported by Sesame. RDQL is the query language of Jena, mentioned in the previous section. SeRQL is Sesame's own query language, which combines

---

<sup>4</sup>See <http://www.ics.forth.gr/>

features of several other query languages along with its own.

The API of Sesame is simple and straightforward, with enough operations to build and manipulate knowledge bases.

## Chapter 4

# Requirements and General Design

### 4.1 Interviews

To understand the NetSim project and the demands on the distributed repository, we interviewed some of the project members. Those we interviewed were the members responsible for distributed collaboration, distributed resource management and execution, distributed repository (our supervisor) and the project leader. The result of the interviews gave an ambiguous view of the distributed repository.

The differentiating views of the repository are understandable, since the interviewees all have disparate needs. Thus their notion about what the repository is and should provide do not always agree. Some functionality that was mentioned, was very specific and in our opinion really up to the programmers who eventually write the NetSim applications to implement. The repository's functionality, we feel, should be of the general kind, one that provides information which these applications can use to build up those very specific functions. Another thing that was confusing was the degree of distribution of the distributed repository, and how dynamic the system should be with respect to joining and leaving of participating nodes. Some argued that in order for certain things to work, we had to have centralised parts, and some others meant that the repository should be completely decentralised.

Among all the various needs of the different parts of NetSim, we could identify requirements on the platform and architecture. This helped us complete our design and choice of underlying platform. Other things that emerged throughout the interviews was which basic operations we must provide to enable those higher level operations the members envision.

## 4.2 Requirements

Here we describe an overview of what we found were the most important requirements, and what we thought was essential for the project. See Appendix A for a complete list and explanation of each requirement.

The requirements upon the distributed repository can be classified into groups:

- Requirements on the underlying platform
- Requirements on the architecture of Carbonara
- Requirements on the functionality of Carbonara
- Requirements on the usability of Carbonara, both from developers view and users view.
- Requirements on the security of Carbonara

Within each group there are a set of requirements that we have found to be the most important for the project, from a view of functionality (what we need at the very least) and from a view of feasibility (what we have time to do). The remaining requirements show what Carbonara aims to be some time in the future.

**Underlying Platform** The most important requirements on the underlying platform is that it should be operating system (OS) independent, open source and implement a well known standard. The goal of NetSim as a whole is to be OS independent, which means that its parts also have to be OS independent. This requirement can be relaxed in the sense that if the application we have chosen to use as platform *aims to be* OS independent, it has fulfilled the requirement. Next, we need to use open source applications, in order to be able to modify the code if needed. And last, use of standards means that we should use applications that implement a standard, so that integration and communication with other systems is simplified. Other requirements are scalability, degree of distribution, administration and operations provided. An active community is also needed to ensure further development of the application.

**Architecture** Concerning the architecture of the distributed repository, two requirements distinguish themselves: modularity and descriptions of resources. The distributed repository must be modular in its structure, both internally and externally against the rest of the NetSim modules. Modularity means ease of replacement of parts and simplified rebuilding. A main feature of the distributed repository is to provide a semantic search. In order to make this possible, resources must be described, i.e. they must have some *metadata* associated to them. Metadata tells which features an object has. We must provide for a

way to register metadata in a predefined language. Other requirements on the architecture are robustness, dynamism (joining and leaving of computers) and replication of data.

**Functionality** The distinguishing factors of the repository are the operations it provides. The most important function is the semantic search. The semantic search enables querying for objects in the repository by describing their features, which are given by the metadata of the objects. Other important functions are storage and retrieval of objects, versioning, tracing, management of access rights and concurrency control.

**Usability** For us, usability means ease of use both for clients using the system and programmers that develop it further. Therefore we must design a well defined user interface, as general as possible. The system must also be well documented, meaning code commentaries together with system overviews and guides. A requirement from the NetSim project was that the system must provide its operations as web accessible services.

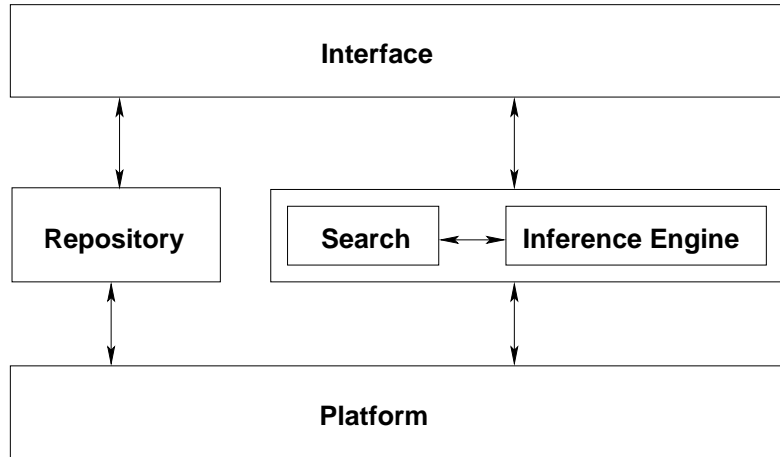
**Security** Although security is of great importance, this project will not have security as a primary concern. Those mechanisms that can be easily implemented or are given by the choice of platform will become part of the repository security mechanisms. The repository should, however, be prepared for future modifications that involve security. Areas that concern security are integrity of data, confidentiality of data, the untrusted environment we operate in, access control, user authentication and consistency in the case of replication.

### 4.3 General Design

During the evolution of the requirements list, a general design was outlined to simplify the search for a suitable platform. The result can be seen in Figure 4.1. It gives a brief description of the system to be developed and its parts.

In the lowest layer lies a platform providing the system with basic functionality. Low level operations such as simple up- and download of resources together with handling and searching of metadata is the responsibility of this component. Although the figure implies that the platform should be a single unit, in reality there is no such restriction. Several different applications that fulfil parts of the requirements for the platform can be integrated in order to supply all needed operations.

The two modules in the middle layer implement higher level operations. These operations complete the basic ones of the platform by configuring and reshaping parameters, adding control mechanisms and error control. The repository module handles storage and retrieval of resources. The search module handles



**Figure 4.1:** General design of the Carbonara distributed repository.

incoming queries and resource descriptions. The search module also interacts with an inference engine to extract more information from the given metadata.

At the top level is the interface, the entity that unifies the middle layer modules. Clients will use this interface to call upon the operations of Carbonara. Therefore it is important that the interface is designed and implemented in such a way that makes it easily adaptable by other applications.

## 4.4 Choice of Platform

We decided to use Globus Toolkit as the platform and OWL along with Jena for the semantic search. Below we discuss why we chose these applications and technologies.

### Distributed Repository

The implementation of peer-to-peer systems we evaluated had not reached a production level. Most projects were still in the research phase. Peer-to-peer systems have a lot of potential because of the dynamic nature of the systems. This dynamic behaviour has a large impact on the security of the system. It can be difficult to obtain a secure system in the multi organisational environment NetSim is designed to operate in. Since they all are in a research phase no real standard is used or even proposed. Pastry and PAST are written in Java so they are not dependent on a specific host environment but the future of these projects is very insecure.

The server based solution was not good because of the magnitude of administration. NetSim may consist of several independent organisations with their own internal administration. To make these organisations cooperate in a virtual organisation manner is difficult because of the static network topology. The static behaviour causes difficulties when organisations are trying to join and leave the network since both servers and clients need to be configured.

The grid solution was finally chosen. The Globus Toolkit is implemented upon a well know standard (OGSI) and is used by several universities and companies. It is also an open source project which is in line with one of the base requirements. The current release of Globus Toolkit is not operative system independent but all new tools are written with the aim of no dependency of operative system. The existing tools are slowly rewritten in the same manner. The Globus Toolkit was also chosen by recommendation of the Department of Systems Modelling because they wanted an evaluation of the platform. The major drawback of this platform is that the standard the implementation is based on is evolving very rapidly. This means that an application is probably not working on two different versions of the toolkit.

## **Metadata Technologies**

The choice of metadata language was dependent on how accurate and controlled the information models and the object descriptions should be. RDF will be the base for expressing metadata, but one of RDFS and OWL must be chosen for designing the information models.

Information models in RDFS will be simple class hierarchies of resource types and properties. There are no other ways to constrain classes or properties than the hierarchy and through domain-range typing of properties. OWL is more complex and expressive and contains terms for constraining both classes and properties, yielding more fine grained control.

After much consideration, OWL was chosen as description language. The object descriptions will be provided by users of NetSim, which may be a source of error. A system with built-in control of descriptions with respect to consistency, both logically and according to the information model, will help keep the descriptions accurate. OWL has this kind of mechanisms, which eventually made us choose it as description language. As this choice was made, the choice of metadata storage system followed up naturally. Jena supports OWL fully and provides many specialised operations for the language. Sesame has only partial support for OWL, even with extension packages. Although Sesame has an appealing architecture, it can not provide an API as convenient as that of Jena. The fact that Jena has both an internal OWL reasoner and the possibility to connect to external reasoners, also contributed to the choice.

# Chapter 5

## Globus Toolkit 3

The Globus Toolkit 3 [17] is a reference implementation of the Open Grid Services Infrastructure [37]. Globus consists of core services, security and a bag of low level tools. The core services manages basic building blocks for applications using Globus. The security component is essential and manages all security issues for a grid application. The disparate bag of tools is used by developers to develop higher level grid services. The bag consists of data management, resource management and information services.

### 5.1 Grid Security Infrastructure

The Globus Grid Security Infrastructure (GSI) is a reference implementation of the OGSA security definition. The security model of GSI is based on Public Key Infrastructure (PKI). The purpose of GSI is to handle the issue of secure communication, single sign-on and security between different organisations with different local security systems.

#### 5.1.1 Authentication

GSI is based on the X.509 identity certificates. A X.509 certificate and an associated private key forms a unique credential set which a grid requester or provider uses for authentication. This credential set is issued by a certificate authority who is trusted by the grid entity requesting the certificate. In order to accept a certificate from a requesting grid entity, the providing grid entity must trust the certificate authority signing the certificate.

The establishment of trust between two different organisations by using X.509 certificates is a sort of lightweight process. Two entities in different organisa-

tions may establish a trusted environment without their entire organisations having arranged a trusted environment. This functionality enables the ability to create virtual organisations with much less effort than using other Public Key Infrastructure (PKI) handlers such as Kerberos.

### 5.1.2 Identity Converting

To enable the connectivity of several heterogeneous security systems, GSI uses gateways. These gateways translate between the X.509-based certificates and other security mechanisms. By using the gateway, an organisation does not need to convert its security system to X.509 based certificates. Instead the local security mechanism and the GSI gateway allows users to authenticate both at the specific organisation and at other organisations from the specified one. These conversions do not require any modification of the local security mechanisms.

### 5.1.3 Delegation and Single Sign-On

In grid systems, a user is often performing tasks at several machines simultaneously. These machines may require resources found on nodes where the user has not signed on to. To acquire the resources, the user then has to sign on at these resources as well. To eliminate this problem GSI introduces X.509 proxy certificates. A proxy is a new certificate issued by the user and not by the certificate authority. The new certificate contains information of the owner identity and a notification that it is a proxy. The lifetime of the proxy certificate is also within the certificate. After that time has expired, the proxy is not accepted by others. Proxies always have a limited lifetimes. A proxy is capable of creating a new proxy certificate and signing them. Figure 5.1 shows a sample proxy delegation scheme.

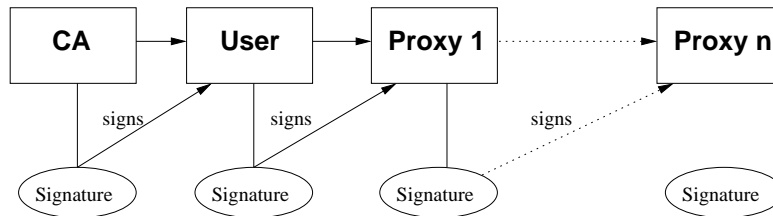


Figure 5.1: Proxy creation

Because no certificate authority is involved in creating proxies these creations are done at any given time.

The proxy certificate is then sent along with the user certificate to the entity providing a service. The proxy certificate is authenticated using the user certificate which in turn is authenticated by the certificate authority.

By using this technique single sign-on is achieved.

#### 5.1.4 Confidential Communication

GSI may run in three different modes. One with no encryption, one with communication integrity and the last mode in full encryption mode.

With no encryption enabled GSI handles the mutual authentication between two entities. When the authentication phase is completed GSI is not used anymore. This reduces the communication overhead by not encrypting and decrypting each message.

When communication integrity is used, GSI secures the message exchange. An eavesdropper may read the communication but not change any of the messages. This security mode adds more overhead than normal message exchange.

In secure communication mode GSI is used to issue a shared key for encryption and decryption. This results in high overhead but the benefit is secure communication. Due to relaxed export-laws from USA it is now allowed to use this technology in GSI.

## 5.2 GridFTP

Grid systems frequently need both large scale transfers of files and access to large amounts of data. These data sets are often geographically distributed. The problem is not new and several systems have been designed to cope with this problem. The downside of these systems are that they use specific and often unpublished protocols for data access. The protocols are often incompatible with others. This leads to a partitioning of the data in a grid system.

To deal with this problem a standard protocol for data transfer on the grid has been proposed[2]. The idea is to use a published and widely used protocol and extend it. For this the File Transfer Protocol (FTP) was chosen. The FTP is a published, widely used protocol with a large knowledge base among developers and a large code base. As FTP is the most common protocol for data transfer and is an IETF standard protocol, it made the choice easier. Finally FTP has support for transfers between two servers with a client acting as a mediator. GridFTP[2] implements this extended protocol.

The most important extensions implemented are Grid Security Infrastructure support, third party control transfer, reliable data transfer and automatic negotiated TCP buffer. Their characteristics are described in the following sections.

## Grid Security Infrastructure support

When accessing data a solid security mechanism is essential. GridFTP uses GSI for authentication, integrity and confidentiality of data access. The features of GSI have been described in Section 5.1.

## Third party controlled transfers

A third party operation allows a user to initiate, transfer and monitor the transfer between two FTP servers. This operation is a common operation in communities handling large datasets in a distributed manner. GSI and its proxy strategy is used to support this type of operation. GSI then handles the authentication on all participants on behalf of the user. Figure 5.2 illustrates this behaviour.

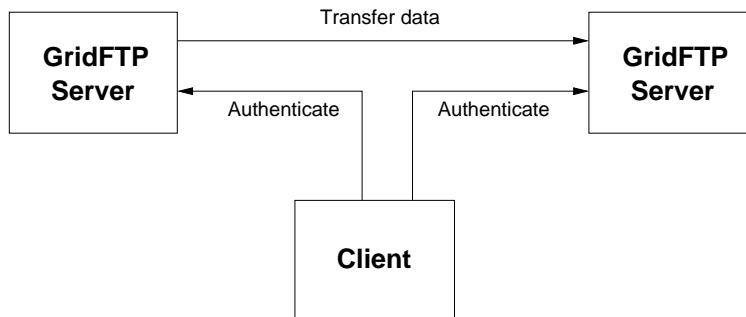


Figure 5.2: Sample third party transfer

## Reliable data transfer

Most applications that manage data need reliable transfers. Recovery mechanisms for handling various network failures, server crashes, etc are needed. GridFTP extends the basic features of the FTP standard for dealing with restartable and reliable data transfers.

## Automatic negotiated TCP buffer

Setting the appropriate TCP buffer can be cumbersome especially for unexperienced users. But with optimal settings for this buffer the impact on transfer performance can be dramatic. GridFTP extends the standard FTP protocol to both accept manual and automatic settings of the TCP buffer for large files and for a large set of small files.

## 5.3 Replica Location Service

To manage replication in a grid system is a difficult task. Data sets may be replicated for reliability and performance reasons. Operations must exist for creating and registering new replicas. The system also requires that replicas can be found. To meet these demands the Replica Location Service (RLS) [10] was developed. RLS provides mechanisms for registering replicas and discovering of registered replicas.

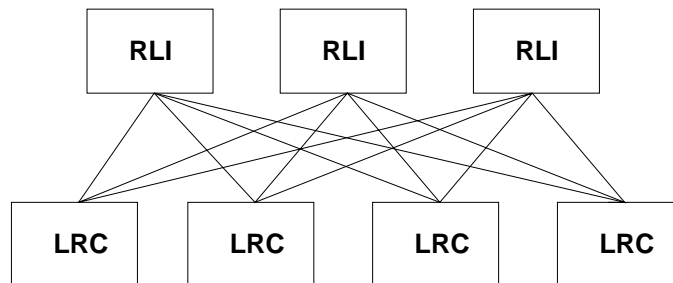
### 5.3.1 Replica Location Service Framework

The RLS framework consists of several building blocks. The most important ones are mentioned below.

- Local Replica Catalogue (LRC)
- Replica Location Index (RLI)
- Soft state update mechanism

A Local Replica Catalogue (LRC) maintains a mapping between logical names and target names. Logical names are unique identifiers for data content. Data content may have one or several physical replicas. Target names are often the physical location of replicas but can also be other logical names.

The higher level Replica Location Index (RLI) handles mappings between logical names and LRCs. A RLI answers queries about mappings within a set of LRCs. By using different configurations of RLI servers different update structures can be constructed. These structures have a variety of performance and reliability characteristics. A sample configuration is found in Figure 5.3.



**Figure 5.3:** Sample Replica Location Service configuration

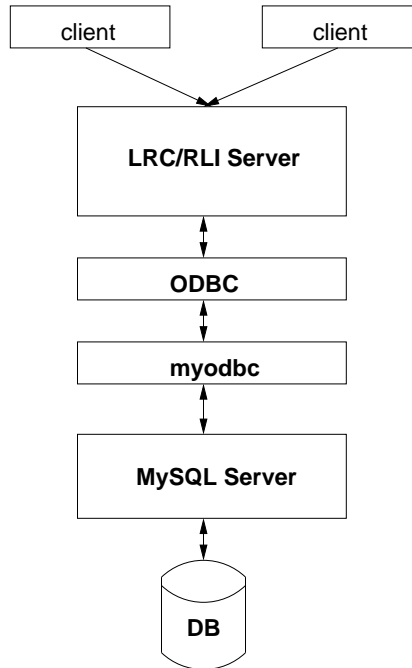
Information updates of RLI components are handled by soft state update protocols. Each LRC sends information about its mappings to zero or more RLIs.

The information in a RLI becomes out of date and has to be refreshed periodically. A failed and recovering RLI is updated during the connected LRCs next update period.

Soft state updates may not always contain all data from a LRC in order to reduce data transfers. Data may also be compressed to reduce data sent. A reduced or compressed set also reduces the storage demands on RLIs.

### 5.3.2 Common Replica Location Service Server

A RLS implementation is made and included in Globus Toolkit 3. RLS consists conceptually of two servers, RLI and LRC, but is implemented as one. The common server may function as a RLI, a LRC or both at the same time. The common server design is shown in Figure. 5.4.



**Figure 5.4:** Common Replica Location Server

The Grid Security Infrastructure is used to support authentication and to connect to the Globus Toolkit. Access lists are configured to allow users to perform Replica Location Service operations such as read from a LRC or write to it. These lists uses user certificates for identification.

The back end of the common server is a relational database. The use of Open Database Connectivity (ODBC) enables usage of several relational databases such as MySQL and PostgreSQL. The RLI and LRC parts of the common server

share the same database.

### 5.3.3 Soft State Updates

Local Replica Catalogues periodically send their state to zero or more Replica Location Indexes. These updates can have different forms.

- Full state update
- Combination of full state updates and incremental state updates
- Compressed state updates using Bloom filters
- Partitioned state updates

Full state updates send all logical names for which a LRC has got mappings, to all RLIs it is configured to update. It is a costly and time consuming approach but very reliable.

A combination of full states updates and incremental state updates reduces the frequency of full updates. Full state updates are performed less frequently and in between only new entries in a LRC is sent to its configured RLIs.

Using a Bloom filter, which is a bit array, reduces data sent from a LRC. Every logical name is hashed using several hash functions. Each result maps to one position in the Bloom filter and these positions are then set. A RLI has to save one Bloom filter for each LRC it handles. When querying a RLI the same hash functions are used and if all functions result in a hit in the Bloom filter the logical name queried is found at that LRC.

Partitioning means that a subset of the logical name set on a LRC is sent to different RLIs. These subsets are normally chosen by using regular expressions in the LRC configuration.

## 5.4 Globus Toolkit and Grid Services

Grid services is an extension of web services. A brief introduction to web services is given before grid services and how Globus uses them is discussed.

### 5.4.1 Web Services

Web services [38] is a technique that supports interoperable network interaction. The technique is not vendor specific which gives disparate systems the ability

to connect. This approach gives the opportunity for different platforms with different hardware and software to interact in a distributed manner. The main part of web services is the interface which clients use to interact with a service. The interface is described by Web Services Description Language (WSDL) [38]. WSDL does not depend on any platform, programming language or paradigm. This gives a system written in C the ability to interoperate with a system written in Java. WSDL is a XML [38] based description language and describes the type of actions a service is providing and how to invoke them.

Clients communicate with web services by message passing. The messages normally use the Simple Object Access Protocol (SOAP) [38] for interaction. These SOAP messages are sent using web related transfer protocols such as HTTP [38] or SMTP [30].

### 5.4.2 Grid Services

Grid services is a technique for bringing service oriented architecture to the grid environment. A web service following the OGSi standard is called a grid service. There are several issues not handled by web services, among others web services do not describe how to create a service nor do they describe the lifetime of the service. They also do not describe how to handle long lived services. These issues are handled by the OGSA in order to provide higher service interoperability. A grid service is transient which a web service can not be. A web service has a life time from the startup of the container to shutdown of the container. A grid service may be created with a predefined life time after which it ceases to exist. The factory pattern for creating instances is used by grid services. One major difference between grid services and web services is that a grid service is stateful which a web service is not. These characteristics results in that a grid service instance may be created for each client accessing the service. A single client will have an instance of its own. This instance can be accessed as long as its life time is not exceeded. The state of the grid service instance is stored between access of the client. With web services, all clients would use the same instance and access data for all clients.

### 5.4.3 Globus Toolkit with Grid Services

Globus Toolkit is implementing several grid services using the OGSi standard. Some components in the bag of tools for the Globus Toolkit have got a grid service interface wrapped around them in order to be used as grid services. These services are file transfer and job management services and are run in a stand alone container provided by the Globus Toolkit. It is also possible to deploy these services on a web container such as Apache Tomcat. Third party grid services are deployed in the same manner. For more information about grid services we suggest consulting the *IBM Developerworks* article archive [1].

# Chapter 6

## The Semantic Tools

In this chapter the metadata languages RDF and OWL will be thoroughly explained together with the metadata storage system Jena.

### 6.1 RDF and OWL

In order to understand OWL, knowledge about RDF is needed. This section will begin by explaining RDF and then proceed to OWL. As stated before, RDF is a language for expressing metadata, i.e. implementing descriptions, whereas OWL is used to construct information models. OWL is an extension of RDFS, which in turn is an extension of RDF. This implies that OWL uses the same syntax as RDF.

#### 6.1.1 RDF

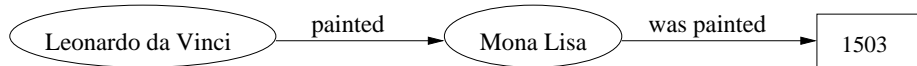
In Section 3.2.1 we briefly explained the foundations of RDF. We will now proceed in explaining RDF further. We mentioned that the basic element of the RDF language is the triple, also called a statement. The triple consists of three parts: the subject, the predicate and the object. Each triple states a fact about a resource, which means that the subject of the statement is the resource being described. The predicate is a property the resource has or is, and finally the object is the value of the property. An example is:

*Mona Lisa was painted 1503.*

Here “*Mona Lisa*” is the subject, a resource. The predicate is “*was painted*” and object is “*1503*”. But *1503* is a literal, which limits it to always act as object.

Resources can also appear as objects in a statement, in which case the statement is describing a *relation* between two resources. The example of Section 3.2.1 is such a statement, it describes the relation between the resources “*Leonardo da Vinci*” and “*Mona Lisa*”.

The official technique for modelling RDF statements is using graphs[25, Ch 2.2]. Resources become nodes in the graph, depicted as ellipses, and properties become directed edges connecting the nodes. Literals are also nodes in the graph, but depicted as rectangles. Figure 6.1 shows the graph of the statements previously mentioned. Since a RDF knowledge base consists of RDF statements, it can also be expressed by a graph comprising of subgraphs for the information models and the instances.



**Figure 6.1:** RDF graph of statements “*Leonardo da Vinci painted Mona Lisa*” and “*Mona Lisa was painted 1503*”.

In Figure 6.1 and the previous examples we have simplified the identifiers of resources and properties to simple names. In reality, however, they are Uniform Resource Identifiers (URIs). RDF uses URIs for naming and identification. In the example below we have used Uniform Resource Names (URNs). They are a subset of URIs used only for naming and have connection to networks. Using URNs the above resources and properties would perhaps become:

```
urn:art/LeonardoDaVinci
urn:art/painted
urn:art/MonaLisa
urn:art/wasPainted
```

Literals do not need a URI. Resources and properties are not required to have URIs that have an actual network location [25, Ch 2.1].

There are several serialisation techniques for RDF, among others Notation3, RDF/XML, and N-triples[31, Ch 3]. We will use RDF/XML, which is the official serialisation technique for RDF. This technique represents RDF using XML syntax. Serialising the above statements in an XML document could possibly look like Listing 6.1.

The first line of Listing 6.1 is the traditional XML declaration line. The second line is the beginning of the `rdf` element, used to enclose RDF content. On lines 3-4, still inside the `rdf` declaration, two namespace prefixes are defined and set to their respective namespace URIs. The namespaces are used to prevent name collision and provide a mean to group related properties, classes or resources. The namespace prefixes (`rdf` and `art`) are used for shorthand writing.

On lines 6-8 a statement is made, that Leonardo da Vinci painted Mona Lisa. Line 6 states the subject, i.e. which resource this statement is about (`urn:`

**Listing 6.1:** Serialisation of RDF statements

```
1 <?xml version="1.0"?>
2 <rdf:RDF
3   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4   xmlns:art="urn:art/">
5
6   <rdf:Description rdf:about="urn:art/LeonardoDaVinci">
7     <art:painter rdf:about="urn:art/MonaLisa"/>
8   </rdf:Description>
9
10  <rdf:Description rdf:about="urn:art/MonaLisa">
11    <art:wasPainted>1503</art:wasPainted>
12  </rdf:Description>
13 </rdf:RDF>
```

`art/LeonardoDaVinci`). Line 7 tells us the predicate (`art:painter`) and the object (`urn:art/MonaLisa`). The property `art:painter` will by a parser be expanded to `urn:art/painter`, i.e. the namespace prefix is replaced by the namespace URI. Line 8 ends the statement.

Lines 10-12 make another statement, that Mona Lisa was painted 1503. Analogue to the previous statement, line 10 says about which resource we are making this statement about. The next line tells us the predicate, and this time the value of the predicate is a literal, 1503. The document and the RDF content are concluded by line 13.

## 6.1.2 OWL

In Section 3.2.3 we gave an overview of OWL. In this section, we will explain OWL more thoroughly. OWL is an extension of RDFS. RDFS provides mechanisms for building information models based on hierarchies of inheritance of classes and properties. In addition to inheritance, RDFS provides mechanisms for defining domain and range for properties. OWL extends RDFS and adds more fine grained control over the design of information models. In OWL, information models can express relations between classes and characteristics of properties. Examples of this is disjointness between classes, cardinality restrictions and type setting of properties. With these possibilities, information models resemble reality more than in RDFS.

Below we will step by step describe a simple information model in OWL. The reader should be aware that we will not show a complete XML serialisation of the information model.

We will use five namespaces in our information model:

**rdf:** The standard RDF syntax namespace.  
Namespace URL: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

- rdfs:** The standard RDFS syntax namespace.  
Namespace URL: <http://www.w3.org/2000/01/rdf-schema#>
- owl:** The standard OWL syntax namespace.  
Namespace URL: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
- xsd:** Standard namespace defining XML datatypes, used by OWL.  
Namespace URL: <http://www.w3.org/2001/XMLSchema#>
- art:** Our example namespace.  
Namespace URL: <urn:art/>

In Listing 6.2 we define two classes, `Person` and `Painter`. The `Person` class is very simple, it is merely a declaration. The `Painter` class is declared to be a subclass of `Person`. If a reasoner is used, it will deduce that any instance of `Painter` is also an instance of `Person`, and add such a statement to the knowledge base. Properties can be ordered in an inheritance tree in the same way as classes. The class definition also states that the class `Painter` is disjoint with another class, `Painting`, yet to be declared. This implies that there must never be an instance belonging to both classes at the same time. If such an instance exists, a reasoner will report it as an inconsistency.

**Listing 6.2:** Declaration of classes `Person` and `Painter`

```

1 <owl:Class rdf:about="urn:art/Person"/>
2
3 <owl:Class rdf:about="urn:art/Painter">
4   <rdfs:subClassOf rdf:resource="urn:art/Person"/>
5   <owl:disjointWith rdf:resource="urn:art/Painting"/>
6 </owl:Class>

```

Next, we define the `Painting` class in Listing 6.3. This class has a cardinality restriction on a property named `wasPainted`, not yet defined. It states that instances of `Painting` may not have more than one value for the property `wasPainted`, otherwise it is an inconsistency. Cardinality restrictions exist for stating a maximum, minimum or exact number of values for a property. The cardinality restriction is constructed in a very special way: `Painting` is declared to be subclass of an anonymous class which has a restriction on said property. Since `Painting` is a subclass of the anonymous class, it will also have the restriction.

OWL provides for a set of different property classes, which give properties characteristics. In Listing 6.4 we define the properties of this simple information model: `wasPainted` and `painted`. The two are declared to be of type `DatatypeProperty` and `ObjectProperty` respectively. A `DatatypeProperty` is a property whose range is a datatype, and an `ObjectProperty` is a property whose range is a resource. In Listing 6.4, lines 22-23, we can see that the range of `wasPainted` is an integer from the namespace `xsd`. On line 28 we can see that the range of `painted` is instances of `Painting`. In most programming languages, domain-range declarations are used to check consistency, i.e. controlling that the subject and object of the property are of the right type. A reasoner, on

**Listing 6.3:** Listing of class Painting

```
7 <owl:Class rdf:about="urn:art/Painting">
8   <rdfs:subClassOf>
9     <owl:Restriction>
10      <owl:maxCardinality
11        rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
12        >1</owl:maxCardinality>
13      <owl:onProperty>
14        <owl:DatatypeProperty rdf:about="urn:art/wasPainted"/>
15      </owl:onProperty>
16    </owl:Restriction>
17  </rdfs:subClassOf>
18 </owl:Class>
19 </rdf:RDF>
```

the other hand, uses the domain-range declarations to deduce new information about the subject and object: the subject may be deduced to belong to the domain class, and the object may be deduced to belong to the range class. In this way, a reasoner that infers over the statements of Listing 6.1 will deduce that LeonardoDaVinci is a Painter and MonaLisa is a Painting.

**Listing 6.4:** Listing of example properties.

```
20 <owl:DatatypeProperty rdf:about="urn:art/wasPainted">
21   <rdfs:domain rdf:resource="urn:art/Painting"/>
22   <rdfs:range
23     rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
24 </owl:DatatypeProperty>
25
26 <owl:ObjectProperty rdf:about="urn:art/painted">
27   <rdfs:domain rdf:resource="urn:art/Painter"/>
28   <rdfs:range rdf:resource="urn:art/Painting"/>
29 </owl:DatatypeProperty>
```

Properties can also be stated to be transitive or symmetric by making the property of the type `owl:TransitiveProperty` or `owl:SymmetricProperty`. This is a signal to a reasoner on what possibilities of inference it has. Properties of type `owl:TransitiveProperty` follow the transitive rule:

*If A is related to B by property P, and B is related to C by property P, then A is related to C by property P.*

By using this rule, the reasoner can add the statement “A is related to C by property P” to the information model. The behaviour of the `owl:SymmetricProperty` is analogous.

OWL is based upon the *Open World Assumption* (OWA), which states that information is not bound to a scope, e.g. a file or a knowledge base. If something can not be proved we do not know its truth status. If a piece of information is not stated in a knowledge base, it does not mean that it does not exist.

It may exist in another knowledge base. According to OWA we can not say that it is false just because it does not exist in our knowledge base. This may cause confusion when designing and using knowledge bases, because they operate under OWA. Regular database systems often operate under the *Closed World Assumption*, which assumes that the database contains all relevant facts, and if a piece of information is not there, it does not exist. The OWA assumption can be a problem when type or property checking instances: if it is stated that all instances of class **A** must have exactly one value for property **P**, it is not inconsistent when an instance of **A** does not have a value for **P**. The declaration for that property value may be located elsewhere. Awareness of these pitfalls makes the design and use of knowledge bases easier in other areas than logics.

In Listing 6.5 the statements from Section 6.1.1 are serialised using the defined information model.

**Listing 6.5:** Listing of instances.

```
1 <art:Painting rdf:about="urn:art/MonaLisa">
2   <art:wasPainted
3     rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
4     >1503</owl:maxCardinality>
5 </art:Painting>
6
7 <art:Painter rdf:about="urn:art/LeonardoDaVinci">
8   <art:painting rdf:resource="urn:art/MonaLisa"/>
9 </art:Painter>
```

We can see that the resources have been given an explicit type using the classes from the above defined information model.

## 6.2 Reasoning

In Section 3.1.4 we defined inferencing or reasoning to be the process of deducing new information from already known information. The deduction follows a set of predefined rules that state what can be inferred if some precondition is satisfied, and what new rules can be triggered. The inferencing is done by an inference engine (also called reasoner) on a knowledge base.

In this section we will describe the foundations of inferencing on RDF based knowledge bases, especially those using OWL. A RDF knowledge base can be expressed as a RDF graph. The reasoner is a tool that expands that graph by discovering new edges between nodes, and sometimes also by discovering new nodes. The reasoner finds these nodes and edges by following the inferencing rules for OWL.

The reasoning over OWL knowledge bases treacherously resembles the type checking and inheritance rules of object oriented programming. But there are fundamental differences. The domain and range of properties are not primarily

for type checking but for inferencing: a resource that has a property can be inferred to be of the domain type of that property. The property value (the object of the statement) can be inferred to be of the range type.

Another example is the cardinality restrictions, mentioned in the previous section. An instance on which there is a minimum or exact cardinality restriction for a property, becomes inconsistent in OWL only when it explicitly violates the restriction. A minimum or exact cardinality restriction that is not fulfilled (i.e. has less than stated number of values for a property) is not inconsistent in OWL, since the OWA says that the information may exist somewhere else, but not here. Only if the number of values exceed the stated limit it is an inconsistency, since the values are explicitly stated, i.e. we know about them.

The examples above clearly illustrate how reasoning in OWL differs from object orientation, and that one must be careful when using OWL and reasoning. It is more of a “backwards” object orientation, and can be misleading when implementing and using a knowledge base. Often, results that are correct from an OWL point of view, are not considered correct by the designer.

By applying the rules of the language, the reasoner can construct new statements to the given knowledge base, completing it further. Information that was not given or visible when constructing the knowledge base can now be deduced and added. This gives us the possibility to see what we have overseen, things that cannot be and things that we did not think of. Inconsistencies can be discovered and corrected. Querying a knowledge base to which a reasoner is connected can yield answers that would not be visible without the reasoner, since the reasoner adds inferred statements.

## 6.3 Jena

Jena[21] is an open source framework for handling and storing metadata. Jena is written in Java and provides an API for creating and manipulating RDF graphs. The languages supported by Jena are among others RDF, RDFS and OWL.

Jena presents an object oriented view of the design and implementation of RDF graphs. All entities and parts of RDF graphs are represented by interfaces. In Jena, a RDF graph is called a *model* and the corresponding interface is called `Model`. There are corresponding interfaces for resources, properties and literals. Through the Jena API, models can be created and metadata can be loaded via files, databases or built up programmatically.

The `Model` interface also makes available a set of simple querying methods, in addition to the more advanced classes belonging to the query engine (see Section 6.4). The querying methods of the `Model` interface are based on triple matching. The triple matching takes a specified subject, predicate and object triple and matches it against all triples in the model. Not all entities of the input triple

must be specified: a non specified entity counts as a wildcard. There are also querying methods via the `Resource` interface, used to model resources. For example, it is possible to query a `Resource` object about all its properties.

Jena provides several internal reasoners of different complexity. They range from the simple `MICRO` reasoner with only domain-range and subclass inference, to a complete `OWL Lite` reasoner. Jena also has the possibility to connect an external reasoner to a knowledge base, via the `DIG` interface mentioned in Section 3.3.1.

It is possible to do consistency checks on knowledge bases to which a reasoner is connected, this is called a *validation*. The validation checks that the information models of the knowledge base are consistent with the chosen language syntax. The validation also controls that the instances of the knowledge base follow the stated structure of the information models. It is also possible to do a *species check* through Jena, i.e. to see which sublanguage of `OWL` the knowledge base conforms to.

## 6.4 RDQL

`RDQL`[33] is a query language for `RDF` based information models. Its syntax resembles that of `SQL`, but its queries are based on `RDF` triples. The queries and the query engine only see metadata as a set of `RDF` triples, and do no inferencing themselves. By connecting a reasoner to an information model it is possible to query over both explicit and implicit statements, since the reasoner adds the implicit statements.

As mentioned earlier in Section 6.1.1, `RDF` triples are modelled as a graph, with resources and literals as nodes in the graph, and properties as edges connecting the nodes. An information model can be a large graph with many nodes and edges built up in several levels. `RDQL` provides a way to define a graph pattern that is matched against the large information model graph. If matches are found, these are returned as a set of variable-value bindings. In `RDQL` the relations between variables (resources) are important, but it is also possible to compare or match literal values of for example properties.

Let us assume that we are interested in querying the model depicted in Figure 6.1, and the question of interest is:

**Question 1** *What relation does Leonardo da Vinci have to Mona Lisa?*

In `RDQL` this would yield the query:

```
SELECT ?relation
Query 1 WHERE (<art:LeonardoDaVinci>, ?relation, <art:MonaLisa>)
        USING art FOR <urn:art/>
```

The pattern that is matched against the graph is (`<art:LeonardoDaVinci>`, `?relation`, `<art:MonaLisa>`), where `<art:LeonardoDaVinci>` is subject, `?relation` is predicate and `<art:MonaLisa>` is object. The question mark in front of `relation` states that it is a variable. The `<>` around the first and last parameters mark that they are URIs, i.e. resources. The URIs will be expanded by the query engine, by using the last row which says that the prefix `art` is a shorthand for `urn:art/`. This means that `<art:LeonardoDaVinci>` is really `urn:art/LeonardoDaVinci`. The query engine will try to match this pattern against the given graph and see which nodes and edges fit into the pattern. The first row tells the query engine which variables to return, in this case only `?relation`. Running the query reveals the answer:

```
urn:art/painted
```

RDQL allows for more complex graph patterns to be built, by combining several triples and reusing variables. The following query exemplifies this:

**Question 2** *Which painting did Leonardo da Vinci paint in 1503?*

```
SELECT ?painting
WHERE (<art:LeonardoDaVinci>, <art:painted>, ?painting),
Query 2 (?painting, <art:wasPainted>,?year)
AND ?year == 1503
USING art FOR <urn:art/>
```

By using the variable `painting` in both patterns, they are connected into a bigger graph. This type of chaining is called a *path*. Running Query 2 gives the answer:

```
urn:art/MonaLisa
```

There are more possibilities with RDQL such as several triple patterns emerging from the same node (variable), filtering and regular expressions. Literals can also be checked for datatypes.

# Chapter 7

## Design

In this chapter we will describe the Carbonara system as it would be in a complete implementation with a fully working platform. We describe how the tools that have been chosen are used and how they interact. We also describe the abstract parts of the system such as resources and operations. We take no consideration to performance or time issues, but describe the system and its behaviour under circumstances with unlimited time and perfectly working tools.

### 7.1 Architecture

In Section 4.3 we described a general architecture of a node in the Carbonara system, with its modules and their internal communication. In this section the architecture is described with respect to the building blocks that have been chosen to form the Carbonara system.

#### 7.1.1 Resources

The distributed repository is populated by what we call resources. A resource is an object of any kind that the user wants to store in the repository. We will here briefly explain what a resource is, but a more detailed description is given in Section 7.2.

A resource consists of four elements. A resource name, the data for the resource, the resource description and the administrative resource description. Resource data and resource description are optional information. The administrative description is set by Carbonara and is only displayed to the user.

The name of the resource, also known as the logical name, has to be unique.

The name has to be a valid URI (Uniform Resource Identifier). The URIs are used to construct namespaces for different organisations.

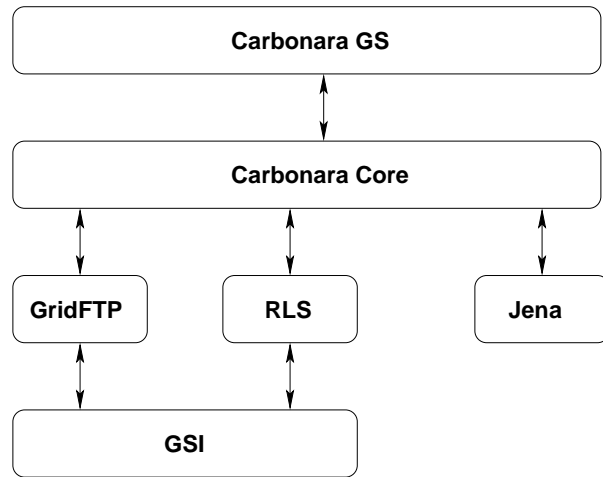
The resource data is the actual information the resource supplies. As an example the resource data for a simulation is the states and the log of the simulation. This resource data of a resource is equivalent to a file and the content of that file.

The resource description is along with the administrative resource description the metadata for a resource. This information is visible to the user and is the base for the semantic search. A resource representation of a computer may contain a description of its computing power, memory and storing devices.

The administrative resource description contains administrative data of a resource, such as ownership and access control.

### 7.1.2 A Carbonara Node

A Carbonara node is an integration of several independent services for querying, location lookup and file transfer. These independent services are integrated in a core module. The core module is also the interface for a Carbonara node. Additional interfaces such as grid services are then wrapped around the core module. In Figure 7.1 a Carbonara node can be seen. The tools within each module of a node will be described below.



**GSI = Grid Security Infrastructure      GS = Grid Service**  
**RLS = Replica Location Service**

**Figure 7.1:** Architecture of a node in the Carbonara system.

**GSI** Grid Security Infrastructure is used by both GridFTP and Replica Location Service to authorise users. These users authenticate themselves with a certificate provided by NetSim's security mechanism[14].

**Jena** Jena [21] provides management of the semantic database as well as inferencing and querying engines for the Carbonara system. The back end of the semantic database is a distributed database management system. Resource and administrative resource descriptions are stored in the semantic database. Jena will return resource names for resources corresponding to queries run in the query engine of Jena.

**RLS** The Replica Location Service [10] handles file locations. Each Carbonara node is running two servers, one Replica Location Index server and one Local Replica Catalogue server. The LRC server handles mapping between a physical location and a resource name. It also sends all resource names to several RLI servers according to a static list of RLI servers. The RLI server of a node gathers information about resource names and the LRC servers they are registered at.

**GridFTP** GridFTP [2] is responsible for file transfer between different nodes. In Carbonara, GridFTP transfers files between the local Carbonara node and physical locations on different GridFTP-servers found by the Replica Location Service.

**Carbonara Core** The Carbonara Core module is the core of the Carbonara system. Carbonara Core integrates GridFTP, RLS, GSI and Jena into a repository system. This module provides an interface for other applications to interact with Carbonara.

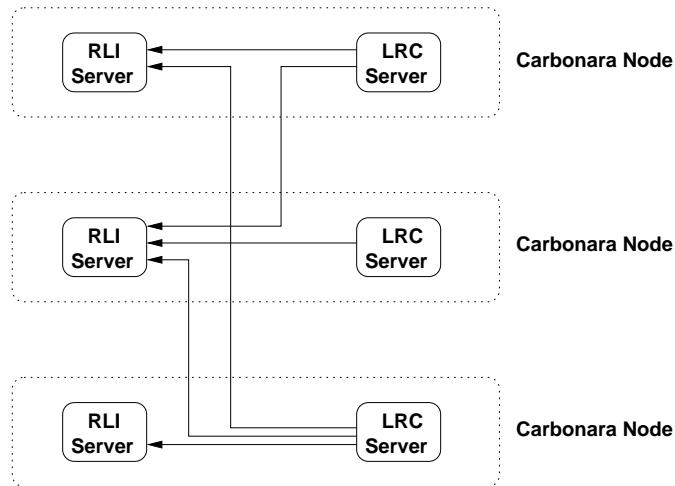
**Carbonara GS** The Carbonara Grid Service module is a wrapper around the Carbonara Core module. It provides the Carbonara system with a grid service interface. The interface provides users with all functionalities found in the Carbonara Core module.

### 7.1.3 The Carbonara Network

The Carbonara system consists of one or several nodes. Each organisation is participating by providing at least one node to the system. Depending on the size of the organisation, they might provide several nodes as well, for instance one for each department. The exact number of users a node can handle depends entirely on the hardware used for each service.

Nodes are connected to each other in a static manner. They connect to each other using the Replica Location Service. As mentioned in Section 7.1.2, a Carbonara node has both a LRC server and a RLI server. The LRC server of a Carbonara node has a list of RLI servers to send updates to. The list is maintained by the administrator of each Carbonara node. Since a Carbonara node has both a LRC server and a RLI server, the list may contain the RLI server on the node itself. The LRC server of a Carbonara node is then sending updates to the RLI server on the same Carbonara node.

A Carbonara node may use the information stored on another Carbonara node if both nodes are connected to the same RLI server. That RLI server can be at any Carbonara node in the system. A Carbonara system is connected if any two Carbonara nodes are connected to one RLI server where that RLI server may reside on any Carbonara node in the system. If this condition is not met, the system is partitioned. A system with all nodes connected is shown in Figure 7.2.



**Figure 7.2:** RLS configuration of a Carbonara network.

If a Carbonara node is connected to all RLI servers in the system the system is fully connected. No single point of failure exists because every two nodes are connected to more than one RLI server. Nodes leaving and joining the system have no effect on resource lookups.

The difference between having a system connected or fully connected is the amount of updates sent to each RLI server. If only a few nodes connect to one RLI server the server handles a small amount of data. If a RLI server is receiving updates from all nodes, it has to handle a large amount of data. The system is faster with fewer connections but more efficient with more connections.

The perfect configuration is a trade off between high availability and efficient updates. A system with few nodes joining and leaving the system should not register many nodes at one RLS server. But a system with the opposite be-

haviour should have a node register at many RLS servers.

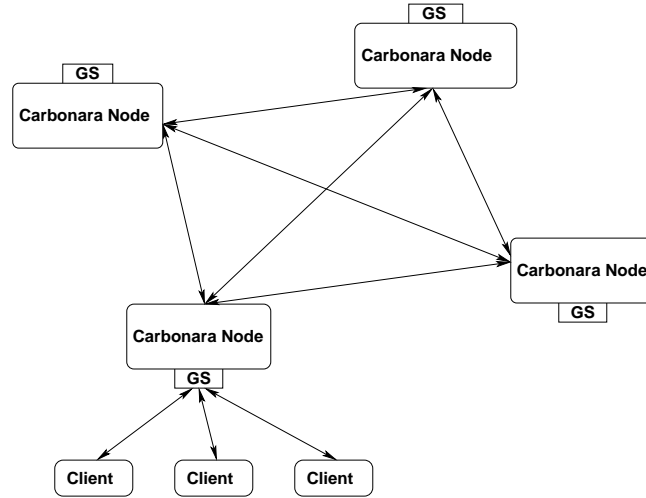


Figure 7.3: Client interacting with a Carbonara network.

Users interact with the distributed repository through a grid service interface. The grid service interface provides the user with all operations available on resources in the distributed repository. The entry point for a user is the local node of the users organisation. The user may be an actual user using a resource manager for exploring the distributed repository or other applications utilising the repository. The client interaction with the distributed repository is found in Figure 7.3.

## 7.2 Resource Metadata

Resource metadata consists of two parts, one administrative and one descriptive. It is used to classify the resource and to reflect resource characteristics. Metadata is also used for internal handling of resources. The metadata is based partly on a predefined, Carbonara specific information model and partly on information models defined by the NetSim users. These information models define available classes and properties. Resource metadata and the information models are stored in a semantic database. A reasoner attached to the semantic database infers new statements about the resources. The inferred statements are also stored in the semantic database. Resource metadata is the foundation of the semantic search.

An important fact to stress is that when resources and resource metadata are discussed, the resource metadata often becomes the resource itself, and the words are used interchangeably. Also the terms *class* and *type* have the same meaning when applied to a resource. Properties are handles for attribute values and are often also denoted as *characteristics* or *attributes*.

## 7.2.1 Information Models

Resource metadata in the Carbonara repository is based partly on a predefined, Carbonara specific, information model and partly on information models defined by the NetSim community. All information models are implemented using OWL. The predefined information model is based on the two root classes `RepositoryObject` and `ClassifyingObject`. A utility class `ACL` (short for *Access Control List*) is used for access control management. An overview and brief description of the root and utility classes of the predefined information model is provided in Table 7.1. They will be more thoroughly described in the following sections.

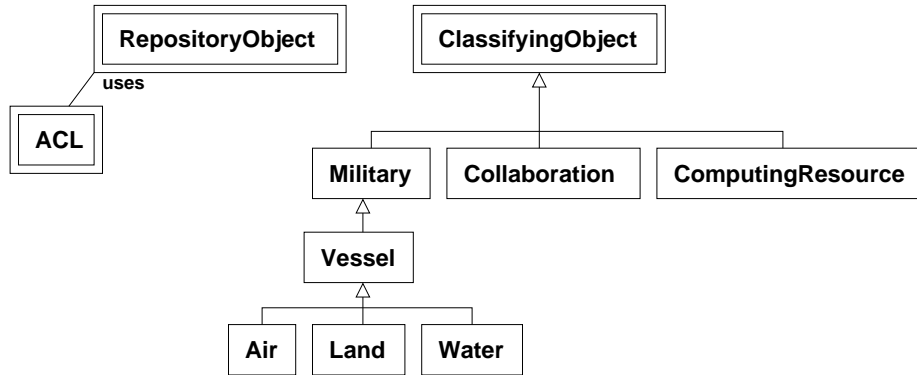
**Table 7.1:** The root classes of the information model, together with the utility class `ACL`.

Class	Description
<code>RepositoryObject</code>	Stores administrative information about the resource.
<code>ClassifyingObject</code>	Stores descriptive information about the resource.
<code>ACL</code>	Stores access rights per user. Utility class used by <code>RepositoryObject</code> .

The administrative part of the predefined information model comprises of the classes `RepositoryObject` and `ACL`, and properties applying to them. The management of resources in the Carbonara system is based on this internal part. Ownership, access control and other administrative attributes are handled through these classes and properties.

The information models defined by the NetSim community are used for resource characteristics. The Carbonara system puts a requirement on these information models: that all defined classes must directly or indirectly inherit the class `ClassifyingObject`, defined in the Carbonara specific information model. In order for the Carbonara system to be able to separate information supplied by the user and information added by the reasoner, this requirement must exist and be fulfilled. Thus, all extensions to the existing set of information models must be designed as subgraphs to the graph with root node `ClassifyingObject`. The internal part can not be extended. An example of the predefined information model together with an extension is depicted in Figure 7.4.

The externally defined information models are allowed to contain all features of the OWL sublanguage that is supported by the used reasoner. There are no other restrictions on these information models, except that all their classes must inherit the `ClassifyingObject` class.



**Figure 7.4:** Example of an extended set of information models. Classes in double frames are predefined and are the base of the information model, all other classes are extensions.

### 7.2.2 Inferencing

The inference engine of the system infers over all metadata that is stored, and adds new statements about resources. Jena has its own built in inference engines of different complexities, but there also exists possibilities to connect an external reasoner to the semantic database. The only requirement on the inference engine is that it supports the language in which the information model is written, in this case OWL.

### 7.2.3 Semantic Database

The semantic database is provided by Jena through the use of a distributed database. Jena uses the distributed database to persistently store the information model and all resource metadata. An inference engine is via Jena connected to the semantic database, and all statements that are inferred by the reasoner are stored in the semantic database. Through Jena, the database processes semantic queries. The database is distributed among all participating nodes.

### 7.2.4 Administrative Information

The administrative resource description is maintained by the Carbonara system. Users can not directly manipulate this information, but have to use the provided operations of the Carbonara interface. The administrative information is sorted under the class `RepositoryObject`. This class contains all administrative properties that are used by the Carbonara system. At all times, each resource will have a subset of these properties set. A list of all the properties of the `RepositoryObject` class can be seen in Table 7.2.

**Table 7.2:** Structure of a `RepositoryObject` instance.

Property	Range Datatype	Description
<code>owner</code>	String	User name of owner. Cardinality=1
<code>acl</code>	ACL	Access control list per user. Cardinality=1..*
<code>ctime</code>	DateTime	Date and time of creation. Cardinality=1
<code>version</code>	float	The version number of this resource. Incremented by 0.1 for each version. Cardinality=1
<code>parent</code>	<code>RepositoryObject</code>	States from which resource this resource is branched. Cardinality=1
<code>mtime</code>	DateTime	Date and time of latest modification. Cardinality=0..1

When a resource is created and for the rest of its existence, the properties `owner`, `acl`, `ctime` and `version` are set. A resource must at all times have an owner, and the owner must have an access control list. The access control lists for each resource are stored through the property `acl`. A detailed explanation of the class `ACL` and how access control is managed is given in Section 7.2.7.

A resource will also at all times have a creation time and a version number, stored in the properties `ctime` and `version`. When a resource is created, the version number will be set to 1.0. At the creation of a new version of a resource, the new version will get the original version number incremented by 0.1. New versions can only be made from already existing resources.

The `parent` property is used in two cases. The first case is when a new version of an existing resource is created. The existing resource then becomes the parent of the new version. The second case is when a new resource is based on another resource, but not strictly a new version. This is called a *branch*. A branch can be used when a new version is not appropriate or in other situations when it is needed to signal inspiration from another resource. The `parent` property is then set to the name of the inspiring resource. The version number of the branch will be set to 1.0. Branches can only be made from already existing resources.

Whenever a resource is modified, the modification time will be stored. The property `mtime` is used for this purpose.

## 7.2.5 Descriptive Information

All resources in the Carbonara system may have a resource description, which can be modified by users who have the proper access rights. The descriptive information is based on the part of the information model that is provided by NetSim. This part defines classes and properties which a resource can be and have. An example of such a class hierarchy can be seen in Figure 7.5.

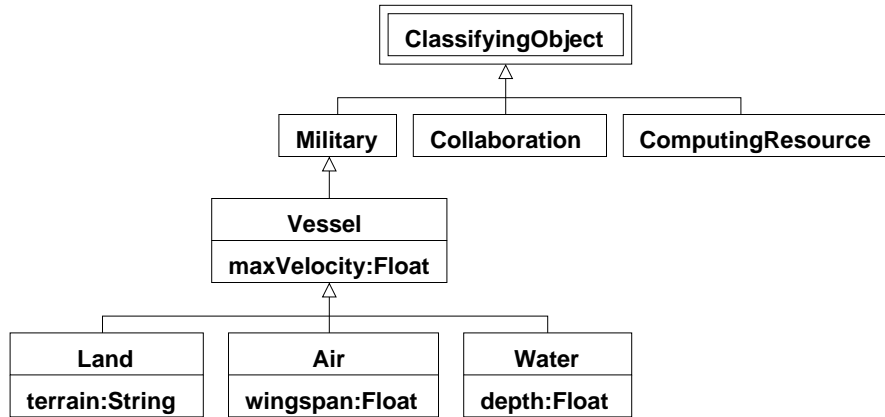


Figure 7.5: Example of a class hierarchy with properties.

The resource description stored as a set of statements in the semantic database. It contains statements that declare the type of the resource, or property-value pairs declaring characteristics of the resource. Internally, the description can also contain other statements which have been inferred by the reasoner. All statements, both declared and inferred, are used when querying the database.

## 7.2.6 User Management

Users do not exist as entities of their own in the Carbonara system. They are represented by a string version of their user name and appear only as property values. Specifically in the `owner` property of `RepositoryObject` (see Section 7.2.4) and the `user` property of `ACL` (see Section 7.2.7). The Carbonara system assumes that users somehow verify their identity to the NetSim system (e.g. by user name and password), and that the NetSim system forwards proper user identification (such as certificates) when performing operations on the Carbonara system.

The Carbonara system supports user identification in the form of certificates. The certificate is created and verified by NetSim's security mechanism. When a user wants to perform an operation on the Carbonara system, its user certificate is forwarded to the Carbonara node performing the operation. The user certificate is then used to verify the user's identity and access rights by matching the certificate user name with the user name in the system.

## 7.2.7 Access Control

To control access to and visibility of all elements of a resource, the Carbonara system makes use of access rights and access control lists.

### Access Rights

There are three different kinds of access rights in the Carbonara system: *read*, *search* and *write*. An overview of their meaning can be seen in Table 7.3. The owner of a resource also has some additional rights.

**Read** Read access on a resource gives the user the right to download resource data. It also gives the user right to view the resource description.

**Search** Search access on a resource gives the user the right to see the resource name as a result of a search. At each query on the semantic database a result set is created, containing the resources matching the query. The user is checked for search access on all resources in the result set. If the user does not have search access on a resource, the resource will be removed from the result set and hence not seen by the user.

**Write** Write access on a resource gives the user the right to upload resource data and modify the resource description. Write access also gives the right to create branches of a resource.

**Owner** The owner of a resource has the right to change access control lists and ownership of the resource. The owner also has the right to create new versions of the resource.

**Table 7.3:** Access rights in the Carbonara system.

Read	Right to download resource data and description. Right to view entire resource description.
Search	Right to view the resource name as a result of a search.
Write	Right to modify resource data and description.
Owner	Right to change access control lists. Right to change owner of a resource. Right to create new versions of a resource.

## Access Control Lists

Access control lists in the Carbonara system are realised through the class `ACL` in the information model. An overview of the structure of an `ACL` instance can be seen in Table 7.4. Each `ACL` instance states which user it applies to and what rights this user has. A resource is connected to instances of `ACL` through the `acl` property of the `RepositoryObject` class. A resource will have an `ACL` per user that has any access rights on it. Changes on access rights can only be made by the owner of the resource.

**Table 7.4:** Structure of an `ACL` instance.

Property	Datatype	Description
<code>user</code>	String	The user to which this access control list applies. Cardinality=1
<code>read</code>	boolean	True if the user has <code>READ</code> access. Cardinality=1
<code>search</code>	boolean	True if the user has <code>SEARCH</code> access. Cardinality=1
<code>write</code>	boolean	True if the user has <code>WRITE</code> access. Cardinality=1

## 7.3 Version Handling

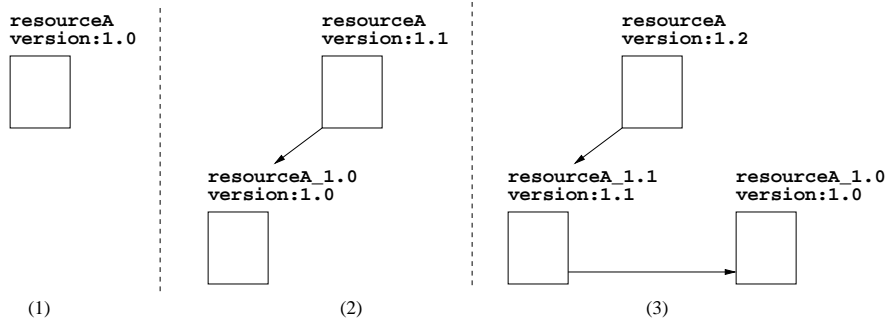
The Carbonara system supports versioning using both version numbers and branches. Branches and versions are very similar operations. Versions may only be added by the owner of a resource, whereas branches can be made by users who have write access on the resource.

### 7.3.1 Versions

A version family is the set of all versions of a resource. Version handling in the Carbonara system is based on that one resource is always representing the family, having the latest version number. The representing resource is physically always the same one. If the family consists of all versions of the resource named `resourceA`, only the representing resource will have that name. All other versions will be named `resourceA+[version number]`. The name `resourceA` is the family name.

When a new version is to be made, a new resource is created and all metadata from the representing resource is copied into that new resource. The new resource will be named `[family name]+[version number]`. The version number of

the representing resource is then incremented, and its **parent** pointer is set to point to the new resource, i.e. the old version. The data of the representing resource is also copied, and the location mappings are changed so that logical names and physical positions are correct with respect to the old version of the resource. This procedure ensures that the most recent version of a resource is the one representing the whole version family and that older versions are only seen when explicitly asked for.



**Figure 7.6:** Versioning

The creation process is depicted in Figure 7.6. In part (1), only one version exists, with version number 1.0. In part (2) a new version has been added. The information in the representing resource has been copied to a new resource, version 1.0. The version number of the representing resource has been incremented to 1.1 and its **parent** pointer is set to version 1.0. In part (3) an additional new version has been created, following the same procedure. The parent of the representing resource is now version 1.1. The version number of the representing resource has increased to 1.2. At all times the version family can be traced by following the **parent** pointer of the resources.

### 7.3.2 Branches

Branches, like versions, have copied metadata and data from the resource they are based on. The differences are on naming, version number and **parent** pointer. The name of the branch will be given by the user creating the branch and must differ from the name of the resource it is based on. The version number of the branch will be set to 1.0 and the **parent** property will be set to the resource it is branched from. An example can be seen in Figure 7.7. In part (1) a branch has been created from resource `resourceA`. In part (2) a new version of both `resourceA` the branch (`branch_of_resourceA`) have been created. The parent pointer of the branch has correctly followed the old version, i.e. version 1.0 of `resourceA`. The name of the branch has also been changed, since a new version of it has been created. The new version of the branch has the family name and its parent pointer is set to the first version of the branch.

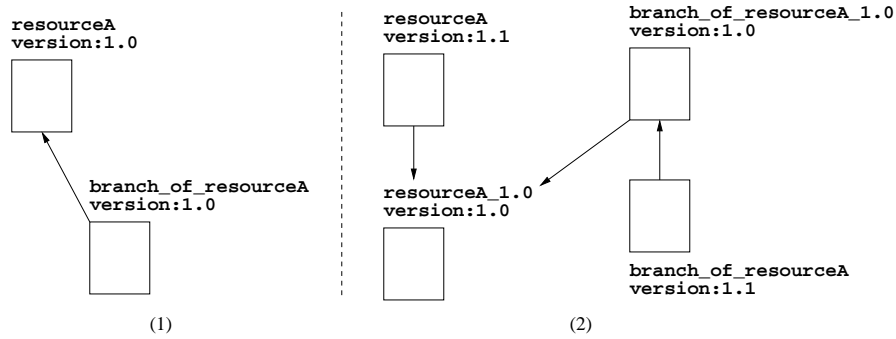


Figure 7.7: Branching

## 7.4 Characteristics of Operations

Carbonara is a repository system with up/download characteristics. A resource can not be modified within the system. It has to be downloaded, modified at the users location and then uploaded to the distributed repository again. This, together with the structure and behaviour of the architecture, reveals special effects when performing operations on the Carbonara system.

### 7.4.1 Resource creation

The creation of a resource contains several steps. The resource name is first registered in the semantic database along with all administrative metadata. The resource description for the resource is then added to the semantic database. When the semantic database is updated, the location of the resource is created. The location is created by the location of the GridFTP server attached to the local Carbonara node and the name of the resource. This location is then registered in RLS. The last phase is to upload the resource data to the location created in the previous phase.

If an error is detected at any phase of the creation process, all information of the resource is removed. This guarantees that a resource will not be inserted if any of its parts are not accepted by the distributed repository.

### 7.4.2 Resource removal

File and database systems normally guarantee that when a file or object is removed the data is unaccessible. The semantic for the Carbonara System is somewhat weaker. Upon resource removal, a resource is removed from the semantic database. All descriptive information of the resource is removed from the database as well. Then all resource data, which is stored on one or several

GridFTP servers, are removed.

If one server is not responding at the time of resource removal, the resource data on that server will not be removed. The result of this behaviour is that resource data may exist after a resource has ceased to exist. That data is only accessible for users manually accessing that specific server. The resource data can not be retrieved using the Carbonara system.

### **7.4.3 Resource update**

A resource is updated by uploading either new resource data or new resource description. The system first checks if the resource exists and if the user has access rights for the operation acquired.

In case of resource description update the old description for the resource is removed and the new resource description is inserted. No mix of resource descriptions is made. This operation is also depending on whether the node responsible for the resource is available or not.

### **7.4.4 Searching**

Queries on the metadata are passed on to Jena, which will process the query and return a result set. If the user performing the request does not have search access on a resource in the result set, the resource will be removed from the set. Hence, the results of a query may only be a subset of all actual results. The results may also be incomplete due to the fact that nodes are not responding at the time of the query and their part of the metadata is unavailable.

Queries are required to be written in RDQL, which is the query language used by Jena.

# Chapter 8

## Implementation

A part of this project was to implement a prototype of the system that was designed. This chapter is dedicated to that prototype.

In Chapter 7 we described the Carbonara system under circumstances with unlimited time and perfectly working tools. This chapter describes the implementation of Carbonara under the time limits we had and the level of functionality of the tools at that time.

We describe which parts of the design that are implemented and the most important classes involved in the functionality of the system. We describe how operations are performed and what their effects are. Finally we present performance tests that were carried out.

### 8.1 Environment

For this project several systems have been used. All code is written in Java with Emacs and Eclipse as developing environments for the core code. A simple GUI was developed using NetBeans. The database management systems used are MySQL and PostgreSQL. The semantic database is handled by Jena with Jena's internal reasoners `MICRO` and `MINI`. The information model was implemented using Protégé. The Globus Toolkit has been used as middleware and Apache Tomcat as a web server. All development and testing have taken place on PCs running Linux. More extensive information about the environment is found in Appendix C.

## 8.2 Prototype Design

This section describes those parts of the design (as described in Chapter 7) that have been implemented in the prototype. We also state limitations and simplifications that have been put on the prototype design.

The implementation of the Carbonara distributed repository consists of five modules, CarbonaraCore, CarbonaraGS, CarbonaraFTP, CarbonaraRLS and CarbonaraSemantic. CarbonaraCore is the core of the repository utilising all other modules except for CarbonaraGS. CarbonaraGS provides a grid service interface for the distributed repository. These classes will be described more thoroughly in other sections.

### Resources

Resources are implemented as described in Chapter 7 with resource description, administrative resource description and resource data. The resource names are valid URIs. However, one restriction has been put upon the naming policy of resources. Due to file system interpretation, resource names are not allowed to contain slashes ('/'). When a client calls for a resource description, it is first written out to a file. The file name is based on the resource name, which if it contains slashes, will be interpreted as a file path by the file system.

### Carbonara Nodes and the Carbonara Network

The Carbonara node is implemented as described in Section 7.1. The Carbonara network, however, is not fully distributed, as was described in the design: the semantic database is a central point in the Carbonara network. Jena provides the semantic database by using a regular database back end. The database back end used by the prototype is singular, hence the Carbonara network has a central semantic database which all nodes use. This is a weakness and a derivation of the original design, but the functionality as experienced from the client side is not affected.

The Carbonara Network expects that all nodes of the system are fully operational at all times. No consideration is taken for handling node failure.

### Information Models

In the design described in Chapter 7, the Carbonara system accepts several information models in addition to the predefined, Carbonara specific information model. The additional models should be created by the NetSim community to be used for resource descriptions.

However, due to a misunderstanding, the concept of multiple information models did not enter the design until late. Hence the prototype is implemented with a single information model. Both the predefined classes as well as the extended classes are contained therein. This is not a serious limitation: everything that can be expressed with multiple information models can also be expressed in a single information model. What is not possible, is to let different organisations or areas define their own information models. They must all use the same information model although they can still define their own structures as part of it.

## **Inferencing and Reasoners**

Because of performance issues the prototype does not use a proper reasoner made for a sublanguage of OWL. The inferencing is reduced to domain/range, subclass and simple cardinality control. The reasoners used are Jena's internal **MINI** and **MICRO** reasoners. Hence not all restrictions defined by the information model can be checked or validated. Since the reasoners only have limited cardinality control, the non-predefined properties of the information model are only allowed a maximum cardinality of 1.

## **Semantic Database**

The semantic database is non-inferencing. The reason for this is that inferencing over persistent models in Jena is too slow to be useful: an operation can take several minutes. To make inferred information available, a work around has been implemented involving an inferencing temporary model. For setting resource description a temporary model is used where local inferencing is done. Local inferencing means that only the schema and the (possibly new) resource description are loaded into the temporary model. The inferred statements of the temporary model are then stored into the semantic database, available for queries. The querying capabilities of the semantic database are not reduced by this work around. The amount of information in the semantic database is only depending on the reasoner and the explicit statements.

## **User Management and Access Control**

Users are implemented as described in the design. They appear only as strings of their user names in the Carbonara system. The user names are used as values for the **owner** property and in the access control lists as values for the **user** property. Access control is also implemented as described in the design.

Carbonara uses certificates for authentication when using Globus Toolkit components such as GridFTP and RLS. Each Carbonara node of the system is currently using one certificate to authenticate itself on another node. When the

NetSim security module is operating each user will have a certificate to access the Carbonara system.

## 8.3 Service Modules

The service modules CarbonaraFTP, CarbonaraRLS and CarbonaraSemantic provide basic functionality around GridFTP, RLS and the semantic database.

CarbonaraFTP is responsible for the low level operations when transferring resource data between different Carbonara nodes. Upload and download methods are implemented in this class.

CarbonaraRLS is handling all interaction with RLS. Physical locations of resources are registered and unregistered in this module. It is also responsible for the lookup of resource data for a resource.

CarbonaraSemantic is the module handling all resource metadata for a Carbonara node. CarbonaraSemantic provides operations for creating and deleting a resource in the semantic database. The access rights for a resource is also handled by this module. Insert, remove and modify metadata in the semantic database for a resource is performed in CarbonaraSemantic.

## 8.4 CarbonaraCore

CarbonaraCore is providing all the functions possible to perform on the distributed repository. The system logic is within this module and the service modules are combined to produce the expected result of an operation.

### 8.4.1 Find Resource

The operation of finding a resource is the search function of the distributed repository. A RDQL query is sent to this method. The service module CarbonaraSemantic is used to query the semantic database for resources matching the query. The semantic database translates the RDQL query into SQL and sends it to the back end database. The result of the query is the names of all resources matching the query. The result list will only contain those resources on which the user has SEARCH right.

## 8.4.2 Lookup Resource Location

The data of a resource may be located at one or several locations. These locations are stored in the Replica Location Service. This method takes the resource name as an in parameter. First a check is made to control if the user is authorised to perform the operations on the resource. The core module then queries the CarbonaraRLS service module which in turn queries RLS. At first, the local LRC server is queried. It returns a list of all locations of resource data for the resource queried. If no locations are returned, all RLI servers that the local LRC server is sending updates to are queried. Each one of them returns a list of LRC servers storing locations of the requested resource. Each remote LRC server is then queried and all locations returned are merged into one list. This list is finally returned.

## 8.4.3 Download Resource

To download resource data the location of the resource and read access for the resource are required. At first, the users read access is checked in the access control list of the resource. The location of the resource is used to connect to the remote computer where the resource data resides. The resource data is then downloaded to the local Carbonara node. The user now has a copy of the resource data.

## 8.4.4 Create Resource

The creation of a resource has one mandatory and two optional phases. The two optional ones depend on the in parameters of this method.

The mandatory phase of a resource creation is to actually create a resource. That means a resource name is registered in the semantic database and an access control list is created for the resource. Preceding the creation, the semantic database is queried for the resource about to be created. If a resource with the same name exists, no action is taken and the method is aborted. It is not possible to create a resource when another resource with the same resource name already exist within the system. If no other resource exists, the resource name is registered and an access control list is created. The owner of the resource is the user creating it. The resource creation is done directly to the semantic database, since no inferencing is needed for the administrative information. The statements are simply inserted into the semantic database.

The first optional phase is to insert the resource description of the resource into the semantic database. This is described in Section 8.4.5. The reason for having this phase optional is that some resources may not have a resource description.

The second optional phase is to upload resource data of the resource on the local

Carbonara node. This is described in Section 8.4.7. The reason for having this optional as well is similar to the above. Not all resources have resource data attached to them. A flow diagram of this operation can be found in Appendix B.3.

### 8.4.5 Set Resource Description

The resource description of the resource is the in parameter of this method. The users access rights are examined in order to prevent unauthorised access. Upon a positive response from the access control check the resource description is loaded into a temporary model with inferencing capabilities. The statements of the temporary model (both the asserted and the inferred) are then stored into the semantic database. If a resource description already exists for the resource it is replaced by the new description. If a user wants to remove the description of a resource an empty resource description is inserted. Then the old description is replaced by the new description which is empty. A flow diagram of this operation is found in Appendix B.1.

### 8.4.6 Get Resource Description

Retrieving a description from the distributed repository requires that the user has **READ** access to the resource of the description. The mandatory control of access rights is made and then the description of the resource is extracted from the semantic database. The extracted description is in the format of an OWL file. This file is then available for the user at its local Carbonara node.

### 8.4.7 Set Resource Data

The operation of setting resource data is divided into two cases depending on if previous resource data exists or not. To find existing resource data, the RLS is queried. If locations are returned in response to a resource name given, resource data for that resource exists.

If no locations for resource data are found, no resource data for the resource exists in the distributed repository. The data is then uploaded to the local Carbonara node and the location is stored in RLS.

If locations are found then resource data for the resource exists. An extensive lookup is performed in RLS by asking all LRC servers for locations of the resource. These locations are formed into a list. The resource data is then uploaded to each location in the list obtained from RLS.

A flow diagram of this operation can be found in Appendix B.2.

### 8.4.8 Remove Resource

The removal of a resource involves all services in the Carbonara system. At first, the user removing the resource must be the owner of that resource. The resource is then removed from the semantic database along with the resource description and access control list. All statements with this resource as subject or object, are removed. The resource is now deleted and can not be retrieved by any means. The resource name is queried in RLS and all locations of the resource data are merged into a list. This list is processed and the resource data at each location is removed and the location is unregistered in RLS.

## 8.5 Carbonara Grid Service

The CarbonaraGS module is a grid service interface wrapping the core module CarbonaraCore. All public methods in CarbonaraCore are accessible through CarbonaraGS.

The distributed repository needs an interface to interact with other modules of NetSim. At first web services was chosen as the interface to use. But since other service modules of the Carbonara system, such as RLS and GridFTP, required grid services to function a grid service interface was used instead.

The CarbonaraGS module is not necessary for the Carbonara system. Other modules of NetSim may integrate the distributed repository. The grid service module is an extra feature which provides easy access for NetSim modules utilising the distributed repository.

A simple client with a graphical user interface has been implemented. The client supports all operations possible through the grid service interface. The client uses the grid service interface to interact with the Carbonara system. A screenshot of the client is shown in Figure 8.1. In the screenshot a query has been submitted to the Carbonara system. All resources matching the query is shown in the top left corner. One resource has been selected (`urn:test:Submarine_10`). The location of resource data is shown in the bottom left corner. The resource description is shown on the right side of the window.

## 8.6 Performance Tests

The tests that were carried out concentrated on measuring performance time for certain operations. The operations were:

**Create resource** Create a resource with resource data and resource description.

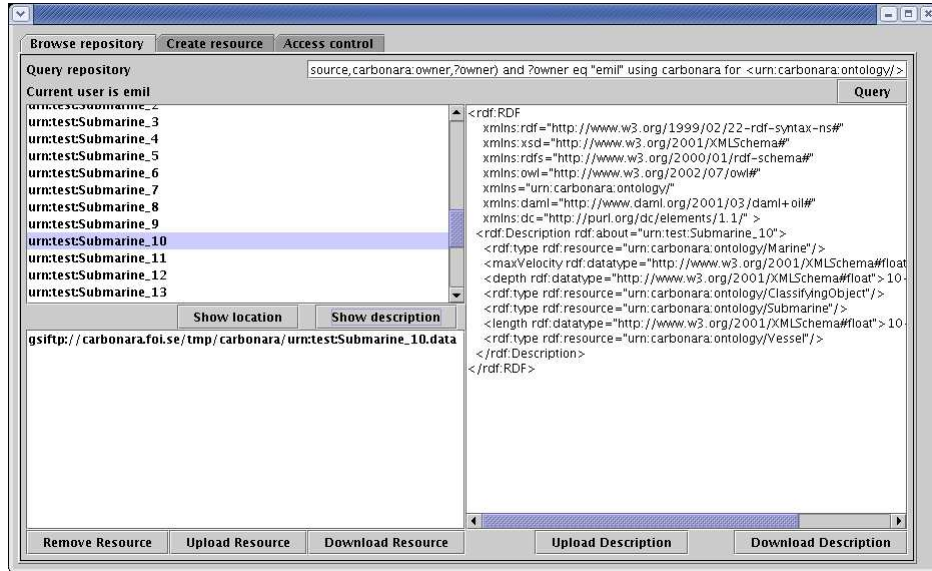


Figure 8.1: Screenshot of the simple client.

**Remove resource** Remove resource, i.e. remove resource data and resource metadata.

**Find resource** Perform a RDQL query.

In the rest of this section we will describe the testbed and explain how each test was carried out together with the results.

### 8.6.1 Testbed

The testbed consisted of two computers, `carbonara.foi.se` and `bolognese.foi.se`, hereafter called `carbonara` and `bolognese`, each offering the Carbonara grid service. Both computers contained equivalent hardware. The semantic database resided on `bolognese`. The client program resided on `carbonara` and contacted one of the grid services using SOAP messages. The client program was text based with no user interaction.

A test consisted of test sets, which in turn consisted of runs. Each test set consisted of 100 runs. For each test set, the system was populated with a certain amount of resources.

The resource descriptions were generated and contained 1-3 property statements. The resource data was the resource description file, with sizes around 630 to 780 bytes.

Between each test, the system was cleaned, i.e. all storages were emptied. The system was not cleaned between test sets.

Result times for sets are average times for the 100 runs. The time measurement was based on system time: we used the difference between the system time before and after the operation was performed. Hence the time measurements do not state the exact time needed for the operation. It is possible that the processor at some rare times has been occupied with operations from other processes which may have affected the results. The reason for this is that the servers were also our work computers, which due to installation problems and lack of time also became our test computers.

## 8.6.2 Creation and Removal

In these tests the client contacted the grid service on `carbonara` only. The tests were carried out as follows. First an initial amount of resources was inserted into the system. The first test inserted 25 resources for which resource data resided on `carbonara`. The second test inserted 100 resources, where 50 had their resource data on `carbonara` and 50 on `bolognese`.

**Table 8.1:** Test results for resource creation and removal at 25 resources in the system.

Set	Resources Locally	Resources Totally	Time for Creation (ms)	Time for Removal (ms)
1	25	25	4493	570
2	25	25	6812	514
3	25	25	11081	495
4	25	25	14547	536
5	25	25	18291	544
6	25	25	20127	506
7	25	25	28769	905

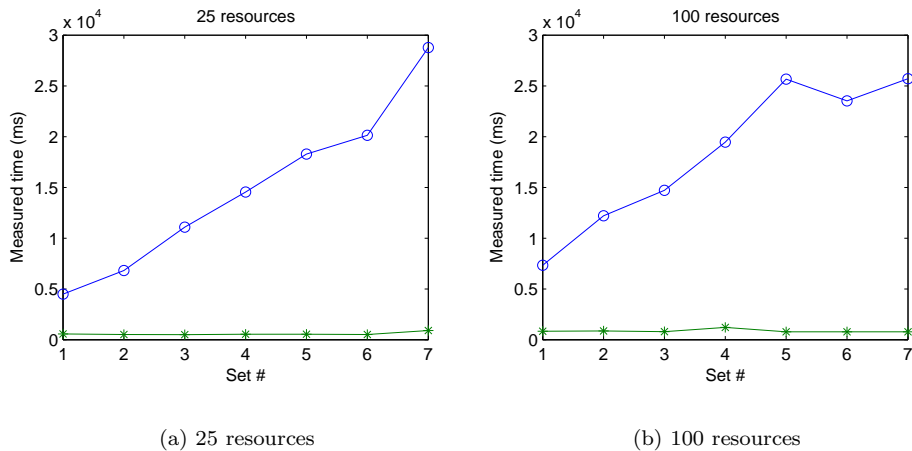
After inserting all needed resources, one resource was removed and recreated in each test run, i.e. 100 times per test set. The system time was taken before and after creation and before and after removal. Execution time was then the difference between start and end times. The average time for the 100 runs was calculated. The overall measured results can be seen in Tables 8.1 and 8.2. Between each set, all inserted resources were removed. Between the two tests, the system was emptied, i.e. all databases cleaned out and all files deleted. The “Resources Locally” column states how many resources have their resource data residing on `carbonara`, the grid service computer that was contacted. The “Resources Totally” column states how many resources there are totally in the system, i.e. on both `carbonara` and `bolognese`.

In Figures 8.2(a) and 8.2(b) the test results are shown as curves. As can be

**Table 8.2:** Test results for resource creation and removal at 100 resources in the system.

Set	Resources Locally	Resources Totally	Time for Creation (ms)	Time for Removal (ms)
1	50	100	7335	826
2	50	100	12210	859
3	50	100	14718	795
4	50	100	19467	1216
5	50	100	25688	784
6	50	100	23529	775
7	50	100	25713	782

seen on both figures, the time for removing a resource from the system is near constant for each test and also much lower than those for creation. The overall average removal time for the first test (Figure 8.2(a)) is 581.4 and for the second test (Figure 8.2(b)) 862.4.

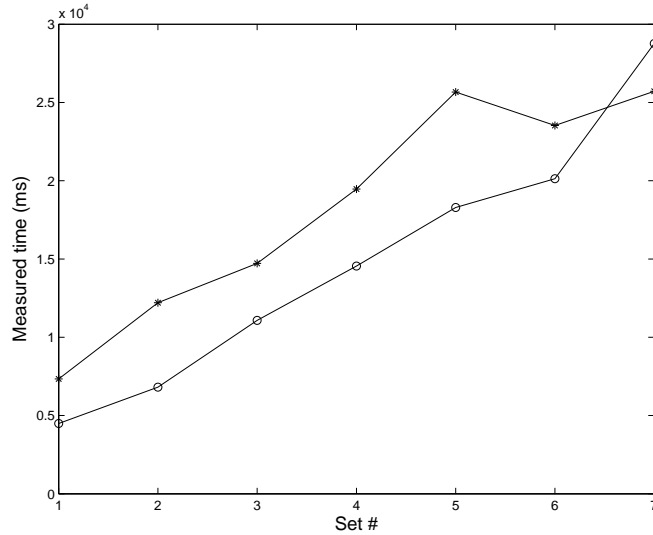


**Figure 8.2:** Graphical overview of test results for resource creation and removal. Upper curve (marked 'o') is creation times, lower curve (marked '\*') is removal times.

When comparing the removal times for the two tests, it can be seen that the removal time increases as the number of resources in the system increases, which is understandable. The reason for the low execution times may be that for the semantic database, the FTP and RLS services, the removal operation is reduced to “delete”, which does not involve many complicated suboperations. Especially for the semantic database, since it is non inferencing, the removal translates to a simple "DELETE FROM" statement in SQL.

The creation times, however, are not constant within the tests and show odd

behaviour. A comparison of creation times of the two tests can be seen in Figure 8.3. Not only are the creation times long, they also increase for each test set. The expected behaviour would have been one similar to the behaviour of the removal times. It seems that the creation time is not only dependent on the number of resources in the system but also on the number of times the operation is performed. The execution time increases as the number of creation operations increase. Both tests show the same behaviour.



**Figure 8.3:** Comparison of resource creation times for 25 and 100 resources in the system. Upper curve (marked '\*') is times for 100 resources in the system, lower curve (marked 'o') is times for 25 resources in the system.

A reason for this may be that one of the Jena models that are used in this operation has redundant or old information that reduces the execution speed. There are two models that are used by this operation. The base model that is connected to the database with no inference and one temporary model with inference. The base model is used by both the creation, removal and query operation. The temporary model is used only when adding resource descriptions, in this case only when creating a resource. Since none of the operations for removal and query show the same odd behaviour, an assumption with high probability is that it is the temporary model that causes the strange pattern.

In the implementation, the temporary model is not reused and it is also emptied of all its statements before the operation returns. Theoretically, the temporary model should not contain any excess information, but it can be that Jena uses a cache or lazy computation that causes information to remain or be reused. This could explain the odd behaviour. No confirmations have been made regarding this theory. There are, however, indications on the Jena development mailing list that some kind of caching is used.

### 8.6.3 Search

In this test the client contacted the grid services on both `bolognese` and `carbonara`. The system had an initial amount of resources which was increased systematically. The same query was processed in all test sets. The test RDQL query was:

```
SELECT ?resource
WHERE (?resource,rdf:type,carbonara:Marine),
(?resource,carbonara:owner,?owner)
AND ?owner eq "emil"
USING carbonara FOR <urn:carbonara:ontology/>
```

In common language the query would translate to “Select all resources of type `Marine`, which are owned by user ‘emil’ ”.

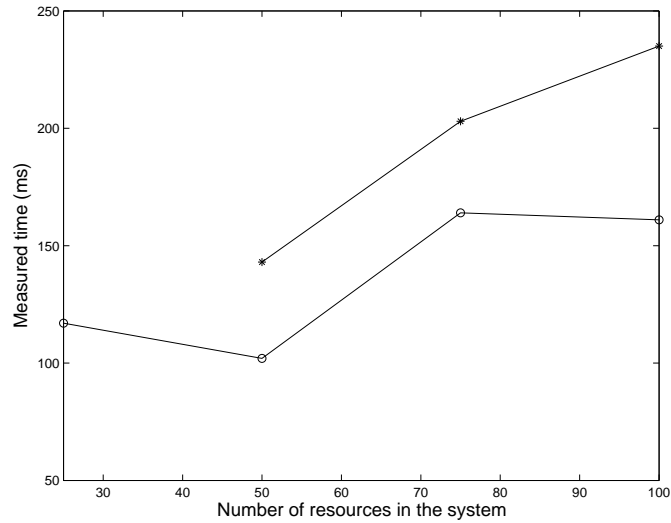
The system time was taken before and after the query was run, the query time was the difference between the two. The numerical results from the test can be seen in Table 8.3. The “Server” column denotes which grid service was contacted. The third and fourth columns denote how many resources had their resource data residing on that server. Total number of resources in the system are the sum of those two numbers.

**Table 8.3:** Test results for resource querying.

Set	Server	Resources on carbonara	Resources on bolognese	Time for Query (ms)
1	bolognese	25	0	117
2	carbonara	25	25	143
3	bolognese	25	25	102
4	carbonara	50	25	203
5	bolognese	50	25	164
6	carbonara	50	50	207
7	bolognese	50	50	161

In Figure 8.4 the results have been plotted. The results have been divided into local and remote calls. By local we mean the grid service that was contacted resided on the same computer as the semantic database, i.e. `bolognese`. As can be seen in the figure, the remote calls take slightly more time than the local calls for the same amount of resources in the system.

This result is expected since there exists a network overhead which adds to the total query time. Generally, we can see that the query times are much lower than those for resource creation and removal. The explanation for this may be that the RDQL queries are translated to SQL, thus using the optimised operations of the DBMS, which are more effective.



**Figure 8.4:** Comparison of query times for local and remote querying. Upper curve (marked '\*') is times for remote queries, lower curve (marked 'o') is times for local.

### 8.6.4 Comments

Since the tests were performed on our work computers, some results differ from the patterns that emerge within a test. This can for example be seen in Figure 8.3, where set 5 in the '\*' marked curve and set 7 in the 'o' marked curve differ from the pattern the rest of the sets make out. Both are higher than would be expected by inspecting the curves. This may have been caused by heavy workload on the computers during that test set.

The tests revealed both expected and surprising results. In the cases for resource removal and querying for resources, the results were explainable and expected. The resource creation, however, showed unexpected behaviour. Already during the implementation, we had experienced that resource creation was slow. But the fact that the execution time is also proportional to the number of times the operation is executed, was astonishing. Unfortunately we have no confirmed explanation for this behaviour, but we recommend further investigation of this matter.

## Chapter 9

# Conclusions and Future Work

Here we present our results and the conclusions drawn from this project together with areas that may be of interest for future development and consideration.

### 9.1 Main Achievements

With this Master's project we have designed and partially implemented a distributed repository called Carbonara. On the basis of interviews and studies we have produced a requirements list. The list fulfils the demands of the NetSim project. Several frameworks have been evaluated in order to find a suitable platform on which the Carbonara system can be implemented. The Globus Toolkit and Jena were chosen as frameworks to design and implement the Carbonara system on. Based on these frameworks we have designed a system that meets the list of requirements. A prototype has been implemented which follows the design and fulfils a subset of the requirements. Simple performance tests were carried out in order to evaluate the system.

### 9.2 Evaluation

In Section 4.2 we described the most important requirements on the Carbonara system from a view of functionality and feasibility. An important matter is how well the design and the prototype meet these requirements.

The requirements for the underlying platform were not completely satisfied. Although Jena is OS independent, the Globus Toolkit is not. The Globus Toolkit

components GridFTP and RLS only operate in \*nix environments. As Globus evolves these dependencies will not remain. Both frameworks follow well known standards and have an active community. The Globus Toolkit configuration does not meet the need for simple administration. Jena does not provide a distributed solution for the semantic database. The scalability of Jena is an unknown factor.

The architecture of the Carbonara system is modular, where each module is replaceable. Because of the design of the RLS component of the Globus Toolkit, the system can not be dynamic. The nodes use static configuration, which also implies the need for an administrator. Replication is neither within the design nor the prototype since there is no common data protection policy for the organisations using NetSim. As in the requirements, each resource has a corresponding resource description.

All functional requirements have been met in the design. The prototype, however, does not meet all these requirements. The operations for versioning and tracing have been prepared but not fully implemented. Concurrency control has been left out of the scope of this project.

As stated in Section 4.2, security requirements have not been of primary concern. Nevertheless, simple access control mechanisms are part of both the design and the prototype. These mechanisms are tightly coupled to the resource description.

The system did not perform in a manner satisfying our expectations. Resource management (creating, removing and modifying resources) took longer time than we expected it to do. Data transfer times were also disappointing, especially at modest file sizes.

The performance of the resource management depends heavily on the performance of the semantic database. Reasoning over persistent knowledge bases is not very efficient in Jena. Especially modifying a resource takes exceptional time. The resource must be fetched, old information removed and new information inserted. These operations are complex and not very effective. To reduce the time consumption we have been forced to simplify the semantic database and use a less complex reasoner. For our implementation, there is no reasoner connected to the semantic database. All reasoning is done in memory based models with only portions of the information loaded. These smaller models are then inferred over and the entire result is stored in the semantic database. The reasoner used for these models is a simple one with only subclass, domain-range and some cardinality check. In this way we could increase performance with only small losses of data and few restrictions.

The security architecture of the Globus Toolkit has some performance drawbacks. This system was originally not constructed for transferring small to medium sized files. The overhead of setting up a secure connection between sender and receiver is time consuming. When transferring large files this overhead becomes only a small portion of the total transfer time. When transferring small to medium sized files the overhead becomes a large portion of total transfer

time.

### 9.3 Comments on the Chosen Platform

**Globus Toolkit** NetSim is a system for information sharing, collaboration and data exchange in a multi organisational environment. Globus is supporting the idea of virtual organisations which is very important in a multi organisational environment. The number of Globus systems running and the amount of funding the project is receiving is making Globus a platform usable for the Carbonara and NetSim projects. The major drawback of Globus is the skill level of the administrator of each node of the system. The administrator has to have good knowledge of Unix systems, user management and configuration of databases. To develop further on the Globus platform the overall knowledge of the system needs to be higher than the present knowledge. More persons in the NetSim project have to increase their knowledge of the system. A new version of Globus is to be released soon. This version will hopefully be more user friendly as the present version is not.

**Jena** Jena is a well functioning framework for RDF metadata. There are many possibilities for using Jena further in the Carbonara system. The drawbacks of Jena are mostly concerning the time consumption: reasoning over a persistent Jena model with the built in OWL Lite reasoner takes a considerable amount of time. Hopefully in the future, Jena will provide a better built in reasoner or will have improved the DIG interface for persistent models, in order to reduce the reasoning time. Jena is a large system, and together with OWL hard to cover fully in just a few months. With more expertise in both areas, Jena will become a powerful tool for future projects. The distribution of the back end database system has also been an issue for this thesis project, and the future may bring a new way of persisting the RDF graphs that can be distributed through Jena.

Installation and use of Jena was never an issue, it was rather the size of the system and its API that made it hard to utilise all available parts of the system.

### 9.4 Future Work

During this project we have only had time to implement parts of the system we designed. During the implementation we have also discovered new aspects that can be considered in order to further develop the Carbonara system. Some of these areas will be described in this section. In reality all aspects of the requirements list (see Appendix A) that we have not fully implemented should be future work. We have chosen to mention the issues that we feel are the most urgent or important at the moment.

## Distributing the Semantic Database

The distribution of the semantic database can be done either by using a distributed database management system or by implementing a communications system between single databases for exchanging result sets. The communication can be of message passing format. Each node can then have a database, on which metadata is stored.

Our implementation of the Carbonara system has a centralised semantic database, although our design was to have a distributed one. To distribute the semantic database or the information contained therein, there exists two approaches:

**Approach 1** Use a distributed database management system (D-DBMS), i.e. distribute the database.

**Approach 2** Use a communications protocol between singular databases, i.e. distribute the information.

In the first approach, the distribution is handled entirely by the D-DBMS, all that is needed is that Jena supports such a system. With the second approach, we mean a system where there are several singular semantic databases, perhaps one per Carbonara node. Each node stores metadata about resources it handles. When a query is issued in the system, the receiving node forwards the query to all nodes, which will reply with a result set. The starting node then puts together all answers and returns them to the user. The communication can be of message passing format, using e.g. SOAP messages. This approach has the drawbacks common for a distributed system: nodes may be down, in which case the query will not return the entire result set.

## Versioning and Tracing

Preparations have been made for both versioning and tracing, but none of the mechanisms are fully implemented. These operations are important to NetSim since the repository will be populated by resources that are frequently updated. In order to handle these updates in a secure manner, a versioning system is essential. In addition, a tracing system is needed to track the ancestors of a resource.

These operations have been partly implemented. The semantic database supports both versioning and tracing. The RLS component of the Globus Toolkit also supports these operations. To achieve full functionality of versioning and tracing, the higher level functions must be implemented in the CarbonaraCore and the functions wrapped in the CarbonaraGS.

## Concurrency Control

Our prototype for the Carbonara system has not been implemented with concurrency control. This is however an area that eventually must be considered. The load on the NetSim system will be of the kind that clients must share resources and that information is read and written frequently. There must exist policies for how to serve clients and how to handle the information modifications during periods of high network load.

Some parts of Jena have concurrency problems. Models can not be modified while being iterated over, i.e. during a read. This is fundamentally a Java `Iterator` issue, which Jena suffers from. Our implementation avoids the problem by letting each client have its own model. But there may be a time when clients need to share models, and concurrency control must be implemented. Either by using queues, locks and critical sections, or any other method for concurrency control. Since it is only the iterators that are the problem, read sections can be made more efficient by copying the information in the iterators to a less sensitive data type. This effectively reduces time spent for reading the model, and also quickly releases the model for other clients to use.

## 9.5 Conclusions

In this Master's project we have developed a foundation for a distributed repository within the NetSim project. The system as a whole is a useful tool for information management, storage and querying. We believe that there is a future for systems like Carbonara.

We can recommend further development of this project as well as more research in this area. The tools we have been using are under constant development, meaning that the performance and usability of these tools will improve. Until then, there are some matters worth noticing:

**Installation of Globus** The Globus Toolkit is unreasonably difficult to install and manage. The expertise needed to set up and administer the system can not be covered by only one person on the department. Until the Java version of the Globus Toolkit is released, it will be time consuming to use the toolkit. Hopefully, the next release of the toolkit will be more user friendly, and easier to handle.

**Reasoning over persistent models** The support for persistent models in Jena is currently poor, although manageable. Jena will in the future have to deal with this, as more and more users demand good performance for persistent models. In the mean time, workarounds will have to do.

Despite the problems concerning the tools used, the project has been very interesting for us. It has contained areas in which we have never worked and we have acquired much new knowledge. When ending this project we truly feel that we have made a contribution to the NetSim project and we hope that the system we have designed will be of use in the future.

# Appendix A

## Requirements List

Through interviews with members of the NetSim project, discussions with our supervisor and among ourselves, a list of comprehensive requirements emerged, which we will define below.

### A.1 Underlying Platform

The platform is the system on which our repository relies. The platform should have some basic features, and provide for low level services, that can be fitted according to our needs.

**Standard** The platform should be either a *de facto* or *de jure* standard, widely used, well developed or under development. It should also have future plans towards features that would enhance the capabilities of the repository.

**Operating System Independence** It is vital that NetSim in the future can operate independently of the underlying OS, but this requirement can be relaxed, if the system found plans to or aims towards becoming OS independent.

**Scalable** As for all peer-to-peer or network applications, the repository should be working equally well with a large number of participants, as with a small number of participants.

**Degree of Distribution** The system should not be centralised, but a complete decentralisation is not necessary. It is acceptable if the system relies on a not so large set of reliable nodes providing the repository functionality.

**Open Source** It is important that we can access the source code, to be able to modify it if needed. Therefore this is a very important requirement.

**Low Need for Administration** We don't want the system to need a full time system administrator. It should be simple enough for a couple of persons to learn, in case something fails, but it should not need frequent administration.

**Low Level Operations** As far as possible, we want to avoid writing our own low level operations. This means that if the platform considered provides relevant operations, it is considered as a great advantage.

**Active Community** An active community is a requirement that ensures the application a future and extensive development.

## A.2 Architecture

The architecture of the repository must be well structured with clearly defined interfaces. If well built, the architecture simplifies the development of the repository and its functionality.

**Modular structure** The repository must be built up in a way that makes it easy to replace and rebuild. This goes for both the repository as a whole and parts of it.

**Robustness** Of course the repository must be robust, meaning a reliable system and well written code.

**Dynamic** It should be fairly easy for service providers to join and leave the system, i.e. the overhead for such an operation must be as low as possible.

**Metadata** Everything that is stored in the repository, whatever it may be, must have a corresponding metadata description to enable the semantic search. This is one of the main features of the repository.

**Replication** Replication depends entirely on the organisations participating in the NetSim network, and how their security policies are designed. With replication, the availability of the repository increases.

## A.3 Functionality

The repository is foremost characterised by the functionality it provides, this distinguishes it from other distributed repositories such as file sharing networks or distributed databases.

**Semantic Search** One of the most important functions is to be able to semantically search for resources in the repository and to receive a list on the resources that satisfied the query.

**Storage and Retrieval of Resources** Uploading and downloading of resources should be easy and according to the access rights of the user. Only the owner of the resources should be able to overwrite them, users may create new resources based on resources they cannot overwrite.

**Versioning** The system should provide for a simple version system.

**Tracing** Resources should “point” back to the resource they are based on, if they are not a new version of the old resource. This provides a means to trace the history of an resource, other than plain versioning.

**Access Rights** Users must have access rights, and be able to set them for resources they create.

**Concurrency Control** A protocol for concurrency control may be needed as users will access resources simultaneously.

## A.4 Usability

We must provide easy-to-use code and API, both for clients using the system, and programmers developing it further.

**Well Defined User Interface** The API must be spartan and general, providing tools for building up more complex operations rather than provide for the complex operations itself. The API must be stripped down to basic building blocks.

**Well Documented System** The system must be well documented. Code must be well commented and a system overview and guide be made.

**Web Service Interface** The NetSim project requires that the repository provides its operations through a web service interface.

## A.5 Security

The repository must both provide for its own security in some areas, and can in some areas also rely on the general NetSim security infrastructure.

**Integrity of Data** Ideally, the integrity of data should be guaranteed, with for example hash sums, but this is not highest priority.

**Confidentiality of Data** Ideally, stored data should be encrypted, preferably with a public key system combined with a symmetric key, but this is not highest priority.

**Untrusted Environment** The repository will be in an environment more or less trusted, which requires certain control over both data, users accessing data and the communication channels in use.

**Access Control** Each resource in the repository must have an Access Control List, defining permitted users/groups and their corresponding access rights.

**User Authentication** We will assume that some sort of user authentication is provided from the NetSim platform, so that users are more or less trusted.

**Consistency** Consistency is needed when replication occurs or for any other reason that demands information modification control.

# Appendix B

## Flow diagrams

### B.1 Set resource description flow diagram

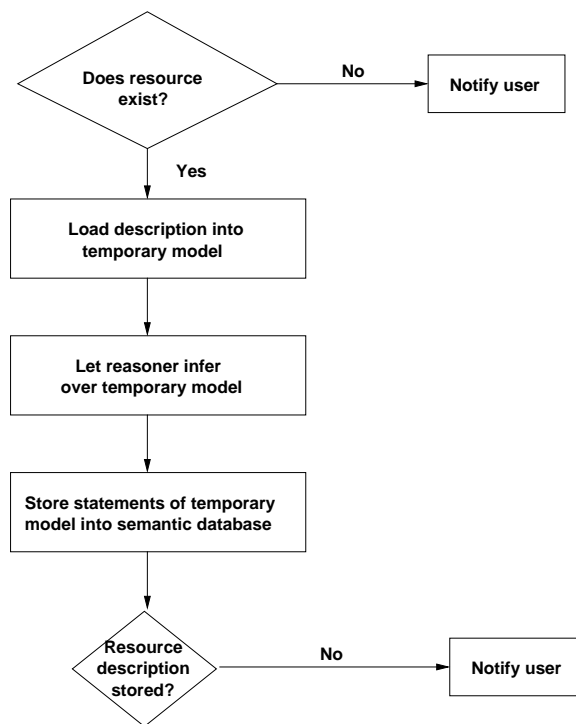


Figure B.1: Flow graph of resource description insert.

## B.2 Set resource data flow diagram

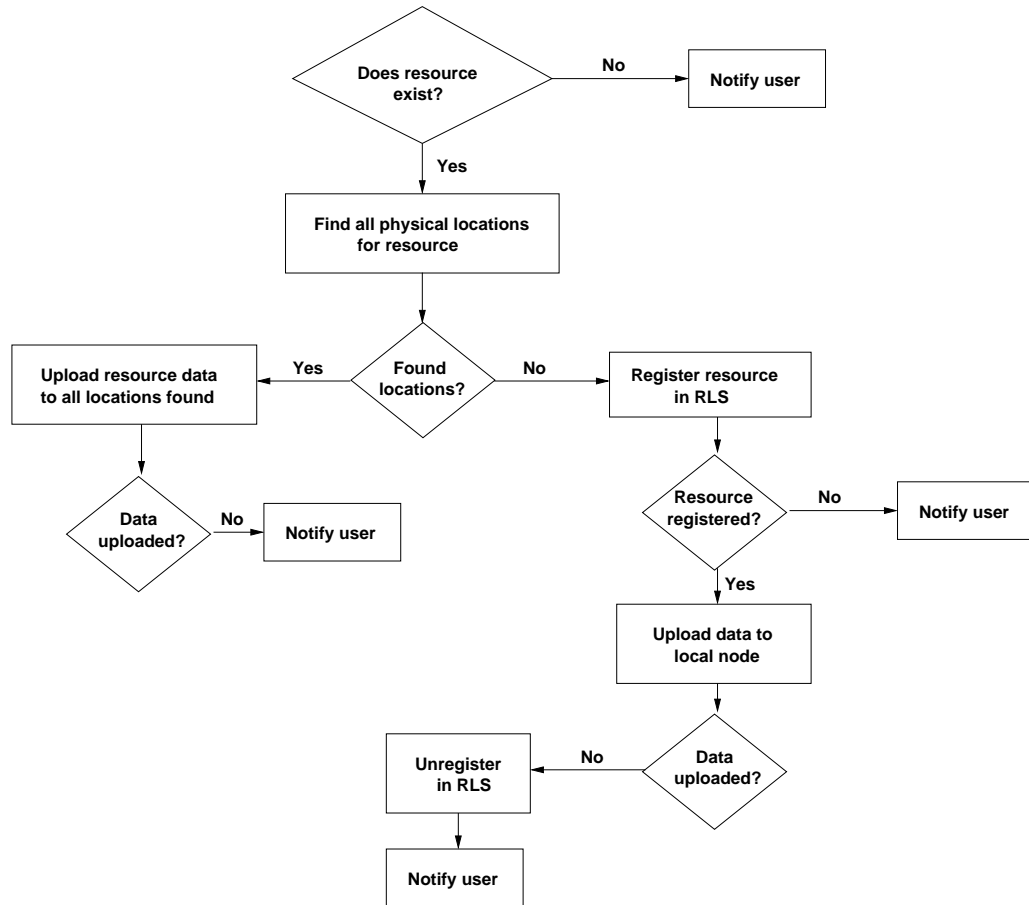


Figure B.2: Flow graph of resource data insert.

### B.3 Create resource flow diagram

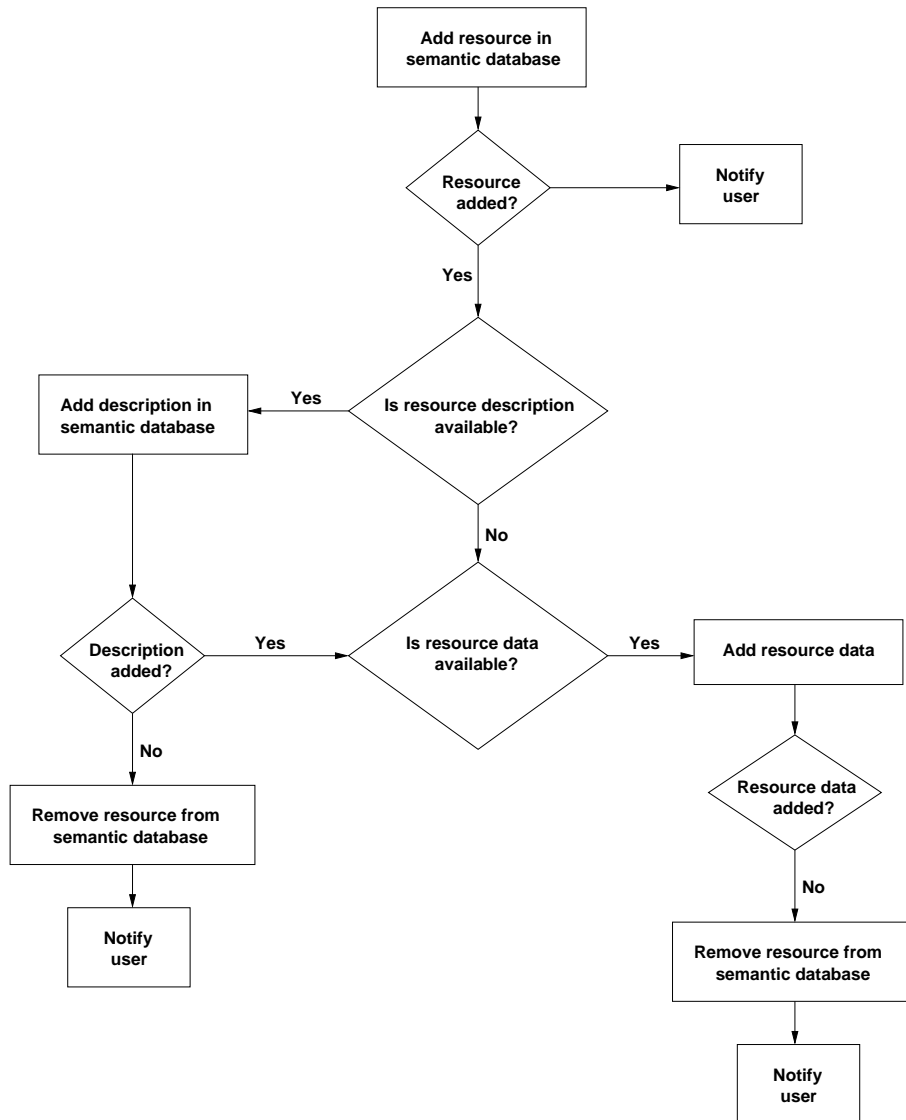


Figure B.3: Flow graph of resource creation.

# Appendix C

## Environment

**Operating System** Linux, specifically the Fedora Core 2 distribution with kernel version 2.6.5.

**Programming Environment** GNU Emacs editor version 21.3.1 and Eclipse IDE version 3.0. NetBeans version 4.0 was used for GUI implementation.

**Programming Language** Java, versions 1.4.2.6 and 1.4.2.4

**Webserver** Apache Tomcat, version 4.1.31.

**Semantic Framework** Jena 2.2 beta 1 with the internal reasoners MICRO and MINI. OWL was used as metadata language. The ontology editor Protégé version 3.0 beta was used for implementing the information model.

**Database Management Systems** Jena has used MySQL distribution 4.1.7, RLS has used PostgreSQL version 7.4.2.

**Grid Framework** Globus Toolkit 3.2.1.

# Appendix D

## Glossary

<b>ACL</b>	Access Control List
<b>AFS</b>	Andrew File System
<b>DBMS</b>	Database Management System
<b>D-DBMS</b>	Distributed Database Management System
<b>DIG</b>	DL Implementation Group
<b>DL</b>	Description Logics
<b>FTP</b>	File Transfer Protocol
<b>GS</b>	Grid Service
<b>GSI</b>	Grid Security Infrastructure
<b>HTTP</b>	Hypertext Markup Language
<b>LRC</b>	Local Replica Catalogue
<b>NetSim</b>	Network Based Modelling and Simulation
<b>OGSA</b>	Open Grid Services Architecture
<b>OGSI</b>	Open Grid Services Infrastructure
<b>OS</b>	Operating System
<b>OWA</b>	Open World Assumption
<b>OWL</b>	Web Ontology Language
<b>P2P</b>	Peer-to-peer
<b>PKI</b>	Public Key Infrastructure
<b>RDF</b>	Resource Description Framework
<b>RDFS</b>	RDF Schema
<b>RDQL</b>	RDF Data Query Language
<b>RLI</b>	Replica Location Index
<b>RLS</b>	Replica Location Service
<b>RQL</b>	RDF Query Language
<b>SeRQL</b>	Sesame RDF Query Language
<b>SMTP</b>	Simple Mail Transfer Protocol
<b>SOAP</b>	Simple Object Access Protocol
<b>SQL</b>	Structured Query Language
<b>URI</b>	Uniform Resource Identifier
<b>URL</b>	Uniform Resource Location
<b>URN</b>	Uniform Resource Name
<b>WSDL</b>	Web Services Description Language
<b>XML</b>	Extensible Markup Language

# Bibliography

- [1] Developerworks Grid Computing Technical Library Archive. <http://www-106.ibm.com/developerworks/views/grid/articles.jsp>, apr 2005.
- [2] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Secure, efficient data transport and replica management for high-performance data-intensive computing. IEEE Mass Storage Conference, 2001.
- [3] BearShare. [www.bearshare.com](http://www.bearshare.com), dec 2004.
- [4] D. Beckett, editor. *RDF/XML Syntax Specification*. World Wide Web Consortium, used version: <http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/>. Latest version: <http://www.w3.org/TR/rdf-syntax-grammar>.
- [5] D. Beckett and J. Grant, editors. *RDF Test Cases*. World Wide Web Consortium, used version: <http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/>. Latest version: <http://www.w3.org/TR/rdf-testcases/>.
- [6] T. Berners-Lee. *Notation3*. World Wide Web Consortium, april 2005. Used version: <http://www.w3.org/DesignIssues/Notation3.html>.
- [7] D. Brickley and R. Guha, editors. *RDF Schema*. World Wide Web Consortium, used version: <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/>. Latest version: <http://www.w3.org/TR/rdf-schema/>.
- [8] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *Proceedings of the First International Semantic Web Conference*, pages 54 – 68, july 2002.
- [9] J. Broekstra and A. Kampmann. Serql: An rdf query and transformation language. In <http://cs.vu.nl/~jbroeks/papers/SerQL.pdf>, august 2004.
- [10] A. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf. Performance and scalability of a replica location service. presented at High Performance Distributed Computing Conference (HPDC-13),Honolulu, HI, 2004.

- [11] DL Implementation Group. <http://dl.kr.org/dig/>, nov 2004.
- [12] DIG Interface. <http://dig.sourceforge.net/>, nov 2004.
- [13] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HOTOS Conf.*, pages 75–80, 2001.
- [14] M. Eklöf, M. García Lozano, F. Moradi, D. Nordqvist, M. Sparf, and J. Ulriksson. Network based modelling and simulation - the architecture. Method report, Swedish Defence Research Agency, dec 2004.
- [15] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2 edition, 2004.
- [16] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In *Global Grid Forum*, june 2002. Latest version: <http://www.globus.org/research/papers/ogsa.pdf>.
- [17] Globus Toolkit 3. <http://www-unix.globus.org/toolkit/>, oct 2004.
- [18] Global Grid Forum. [www.gridforum.org](http://www.gridforum.org), april 2005.
- [19] T. Gruber. What is an ontology? <http://www-ksl.stanford.edu/kst/what-is-an-ontology.html>, march 2005.
- [20] J. Heflin, editor. *OWL Use Cases and Requirements*. World Wide Web Consortium, used version: <http://www.w3.org/TR/2004/REC-webont-req-20040210>. Latest version: <http://www.w3.org/TR/webont-req>.
- [21] Jena - a semantic web framework for java. <http://www.jena.sourceforge.net>, nov 2004.
- [22] G. Karvounarakis, S. Alexaki, V. Christofides, D. Plexousakis, and M. Schol. Rql: A declarative query language for rdf. In *Proceedings fo the 11th International World Wide Web Conference*, may 2002.
- [23] T. Klingberg and R. Manfredi. Gnutella 0.6. [http://rfc-gnutella.sourceforge.net/src/rfc-0\\_6-draft.html](http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html), dec 2004.
- [24] LimeWire. [www.limewire.com](http://www.limewire.com), dec 2004.
- [25] F. Manola and E. Miller, editors. *RDF Primer*. World Wide Web Consortium, used version: <http://w3.org/TR/2004/REC-rdf-primer-20040210>. Latest version: <http://w3.org/TR/rdf-primer>.
- [26] D. L. McGuinness and F. van Harmelen, editors. *OWL Overview*. World Wide Web Consortium, used version: <http://www.w3.org/TR/2004/REC-owl-features-2004-02-10>. Latest version: <http://www.w3.org/TR/owl-features>.

- [27] D. L. McGuinness, C. Welty, and M. K. Smith, editors. *OWL Guide*. World Wide Web Consortium, used version: <http://www.w3.org/TR/2004/REC-owl-guide-20040210/>. Latest version: <http://www.w3.org/TR/owl-guide/>.
- [28] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith. Andrew: a distributed personal computing environment. In *Communications of the ACM*, volume 29, 1986.
- [29] OpenAFS. <http://www.openafs.org>, oct 2004.
- [30] J. B. Postel. RFC 821 Simple Mail Transfer Protocol. <http://www.rfc-editor.org/rfc/rfc821.txt>, mar 2005.
- [31] S. Powers. *Practical RDF*. O'Reilly, 2003.
- [32] Racer. <http://www.sts.tu-harburg.de/~r.f.moeller/racer/>, nov 2004.
- [33] RDQL - a query language for RDF. <http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/>, nov 2004.
- [34] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.
- [35] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. In *IEEE Transactions on Computers*, volume 39, pages 447–459, apr 1990.
- [36] H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index structures and algorithms for querying distributed rdf repositories. In *Proceedings of the 13th international conference on World Wide Web*, pages 631 – 639, may 2004.
- [37] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling. Open Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum, june 2003. Global Grid Forum Draft Recommendation.
- [38] World Wide Web Consortium. [www.w3.org](http://www.w3.org), mar 2005.
- [39] Wikipedia definition of *inference*. <http://en.wikipedia.org/wiki/Inference>, march 2005.
- [40] Wikipedia definition of *inference engine*. [http://en.wikipedia.org/wiki/Inference\\_engine](http://en.wikipedia.org/wiki/Inference_engine), march 2005.
- [41] Wikipedia definition of *knowledge base*. [http://en.wikipedia.org/wiki/Knowledge\\_base](http://en.wikipedia.org/wiki/Knowledge_base), march 2005.
- [42] Wikipedia definition of *metadata*. [http://en.wikipedia.org/wiki/Metadata\\_\(computing\)](http://en.wikipedia.org/wiki/Metadata_(computing)), march 2005.

[43] Wikipedia definition of *ontology*. [http://en.wikipedia.org/wiki/Ontology\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Ontology_(computer_science)), march 2005.