A hybrid peer-to-peer system with a schema based routing strategy

by

Erietta Liarou

A thesis submitted in partial fulfillment of the requirements for the degree of

Electronic and Computer Engineering

Technical University of Crete
Department of Electronic and Computer Engineering

Intelligent Systems Laboratory

Abstract

During the last years, the area of peer-to-peer systems has attracted much interest in the research community. Peer-to-peer technology has become popular mainly through file sharing applications such as Napster, Gnutella and KazaA. At the same time, the amount of available information on the Web is growing, so its organization in a semantic way becomes imperative. The combination of Semantic Web and peer-to-peer technologies will probably provide accurate data retrieval and efficient search. For peer-to-peer environments, metadata are absolutely crucial in order to describe the resources managed by the peers. Schema-based peer-to-peer networks have a number of important advantages over previous simpler peer-to-peer networks.

This dissertation presents the design and development of a schema-based hybrid peer-to-peer file sharing application. Our system supports the existence of different metadata vocabularies between peers and also enables semantic interoperability by providing translation between the different vocabularies. We also modify the hybrid peer-to-peer model. In addition to the traditional distinction between peers, as server or client, we introduce a new distinction between client-peers. It is known that the available bandwidth, the storage space and the processing power vary between computer systems that participate in a distributed system. Thus, in our approach the more powerful client-peers (volunteer peers) remove load from the server, by sacrificing part of their system resources, while the less powerful client-peers (normal peers) participate in the network having the role of the traditional client-peer in a hybrid peer-to-peer system. Finally, the system contains a fault-tolerance mechanism that guarantees connectivity, when nodes of the network fail or leave silently.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The term "peer-to-peer" refers to a class of systems and applications that function in a decentralized way as to achieve share of distributed resources. The area of peer-to-peer systems attracts much interest in the network community, and has become popular through some file sharing applications such as Napster [30], Gnutella [17] and KazaA [22]. At the same time, the amount of available information on the Web is huge and the requirement of its organization, in a semantic way becomes masterful. The combination of Semantic Web and peer-to-peer technologies will probably provide accurate data retrieval and efficient search.

RDF-based peer-to-peer networks can advance the simple peer-to-peer networks like Napster and Gnutella and the more sophisticated ones, based on distributed indices such as CAN [35] and CHORD [21]. RDF-based peer-to-peer networks allow complex description of resources and searching of resources is based on meta-information. This dissertation presents the development of a schema-based hybrid peer-to-peer system, that functions as a resource searing application.

1.1 Overview

A taxonomy of computer systems classifies them into *centralized* and *distributed* systems [28]. Distributed systems can be further classified into the *client-server* model and the *peer-to-peer* model (P2P). The client-server model can be *flat* or *hierarchical.* In the flat model all clients communicate with a single server, while in the hierarchical model, the servers of one level are acting as clients to higher level servers. The peer-to-peer model can be *pure*, *hybrid* or *super-peer* [49]. In pure peer-to-peer systems, there does not exist a centralized server, but all peers have equal roles and responsibilities, while in a hybrid model a centralized server and clients exist. The super-peer model is an intermediate solution, in which super-peer nodes act like peers of pure peer-to-peer systems, but also they are connected with clients in a centralized way. Of all these classes of computer systems, the most popular are the peer-to-peer.

The subject of "peer-to-peer" attracts much interest in the network community. Peer-to-peer systems have emerged as a popular way to share huge volumes of data, for example the system Napster provides support for music sharing on the Web. Napster, Gnutella and KazaA are popular examples of peer-to-peer *file sharing* networks. Another category of peer-to-peer systems is based on *distributed* task execution. In these systems, a central server splits a large task into small independent subtasks and forwards each part to individuals computers. Each computer sends the results back to the server, and then a new job is allocated to the client. In this way, individuals nodes execute work in parallel, so the total task is executed more efficiently and faster. An example of this class of peer-to-peer networks is the system SETI@HOME [41]. Moreover, there are *collaborative* peer-to-peer systems, in which users collaborate in real time. These systems include applications such as games, chat, file sharing and APIs that allow software developers to build their own applications or extend the existing ones. Groove [34] is a system of this category. Finally, there are peer-to-peer systems that belong to the category *platforms*, as for example the JXTA [4] and the .NET [5]. Under the term platform it is meant an infrastructure that enables other applications or other systems to run in a P2P fashion.

Another taxonomy of peer-to-peer systems [28], classifies the P2P networks to these that are simple such as Napster [30] and Gnutella [17], to these that are more sophisticated, using distributed hash tables, like CAN [35] and CHORD [21], and finally to these systems that allow schema description of resources and provide complex queries using metadata, instead of simple keyword-based searches. The Edutella project [44] belongs to the last category. The schema-based networks combines Semantic Web and peer-to-peer technologies in order to make distributed learning repositories possible and useful. These networks rely on the description of resources, using metadata vocabularies. In this way, the information is no longer organized in hypertext like structures, but is organized and stored in a semantic way. The schema-based peer-to-peer networks provide more efficient and accurate search in the Web.

In this dissertation we design and implement an RDF-based, hybrid peer-to-peer resource sharing application. Our system supports the existence of different vocabularies between peers and also enables semantic interoperability between them, since it provides translation between the different metadata vocabularies. We also modify the hybrid peer-to-peer model. In addition to the traditional distinction between peers, as server or client, we introduce a new distinction between client-peers. It is known that the available bandwidth, the storage space and the processing power vary between computer users that participate in a file sharing application. Thus, in our approach the more powerful client peers remove load from the server, sacrificing part of their system resources, while the less powerful clients participate in the network having the role of the traditional client in a hybrid peer-to-peer system. Moreover, the system contains a fault-tolerance mechanism that guarantees connectivity, when nodes of the network fail or leave silently.

The users of the resource sharing application are able to use a number of functionalities. They can:

- Define their own metadata vocabulary.
- Publish files of their computer, describing them by metadata, so other users may see and request these.
- Remove already published files.
- Update the description of already published files.
- Query the system in order to find matching resources owned by other users.

1.2 Organization of the dissertation

This dissertation is organized as follows. In Chapter 2 we briefly discuss alternative peer-to-peer architectures and some well-known, representative networks of these architectures. In the same chapter we also present the necessary background for our system. In Chapter 3 we study the architecture of our system and how the desired functionalities can efficiently be supported using such an architecture. In Chapter 4 we make a discussion about the schema that different types of peers use, and then we analyze in more detail the network functionalities of the system and how these functionalities are executed in a schema-based routing way. We describe the possible scenarios of our resource sharing application. Finally, in Chapter 5 we present our conclusions and future work possibilities.

Chapter 2

Related work and Background

In this chapter we discuss some alternative peer-to-peer architectures and the related systems that support these. We also provide the necessary background for this thesis.

2.1 Related work

In the following subsections we present a taxonomy of computer systems from the peer-to-peer perspective. We also classify the peer-to-peer systems in terms of their application domain. We briefly discuss some alternative architectures for peer-to-peer systems and the some well-known, representative networks of these architectures.

2.1.1 A Taxonomy of computer systems

The computer systems can be classified into *centralized* and *distributed*. Centralized systems represent single-unit solutions, including single- and multi-processor machines, as well as high-end machines, such as supercomputers and mainframes. Distributed systems are those in which their components computers are dispread and communicate in order to exchange information, by passing messages. Distributed systems can be classified into *client-server* model and the *peer-to-peer* model (P2P). The client-server model can be *flat* or *hierarchical*. In the flat model all clients communicates with a single server, while in the hierarchical model, the servers of one level are acting as clients to higher level servers. The peer-to-peer model can be *pure*, *hybrid* or *super-peer*. In the pure peer-to-peer systems, there does not exist a centralized server, but all peers have equal roles and responsibilities, while in a hybrid model a centralized server and clients exist. The super-peer model is an intermediate solution, in which super-peer nodes act like peers of pure peer-to-peer systems, but also they are connected with clients in a centralized way. The taxonomy of computer systems discusses above is shown

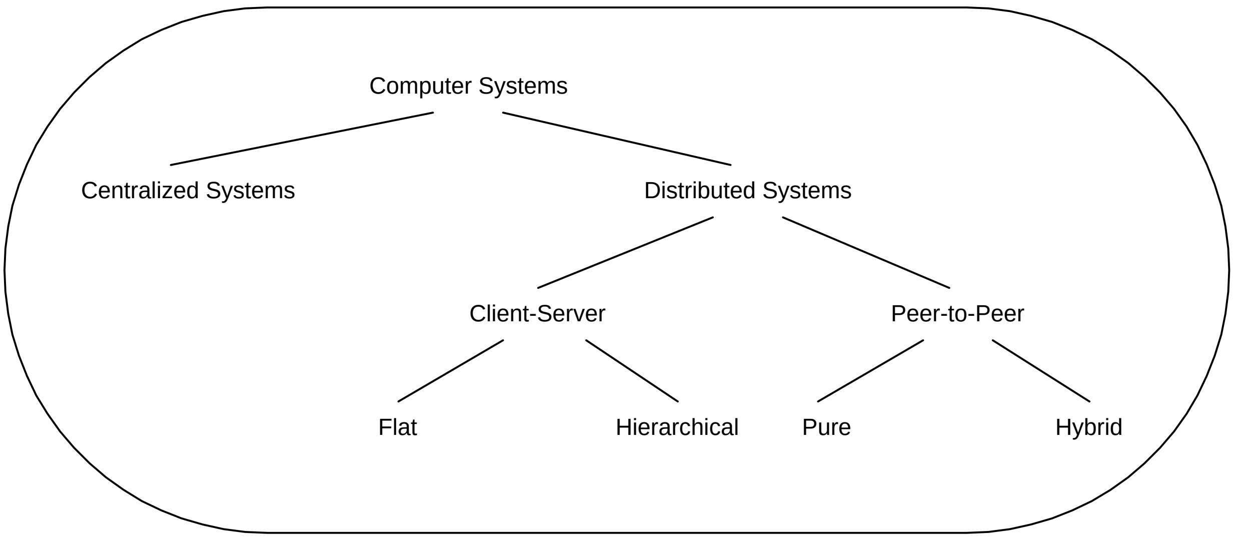


Figure 2.1: A taxonomy of computer systems architectures

in Figure 2.1. More details on taxonomy of computer systems and mainly on peer-to-peer systems are available in [28].

2.1.2 A Taxonomy of Peer-to-Peer systems

A peer-to-peer (P2P) computer network is a type of network in which each workstation has equivalent capabilities and responsibilities, namely it does not have fixed *clients* and *servers*, but a number of *peers* that function as both clients and servers. This type of network differs from the *client-server* model, in which each node has specific role, whether as client or as server.

In Figure 2.2 we present a taxonomy of P2P systems in terms of their application domain. Popular examples of P2P systems are *file sharing* networks such as Napster [30], Gnutella [17], Freenet [16] and Kazaa [22]. The peers of these networks know just a reference to a file and then request it and retrieve it. Another category of peer-to-peer systems is the *distributed computing.* A central server splits a computational problem into small independent parts and forwards each part to individuals computers, which are equipped with the client software. The client software sends the results back to the server, and then a new job is allocated to the client. The system SETI@HOME [41] belong to this category. Also, there are collaborative peer-to-peer systems. *Collaborative* P2P applications allow people on the Internet to meet and work together. Groove is such a system that includes applications such as chat, file sharing, and bulletin boards. It also includes programming libraries and APIs that allow software developers to build their own applications or extend the existing ones. Finally, there are P2P systems that belong to the category *platforms*, for example the JXTA [4] and the .NET [5]. Under the term platform it is meant an infrastructure that

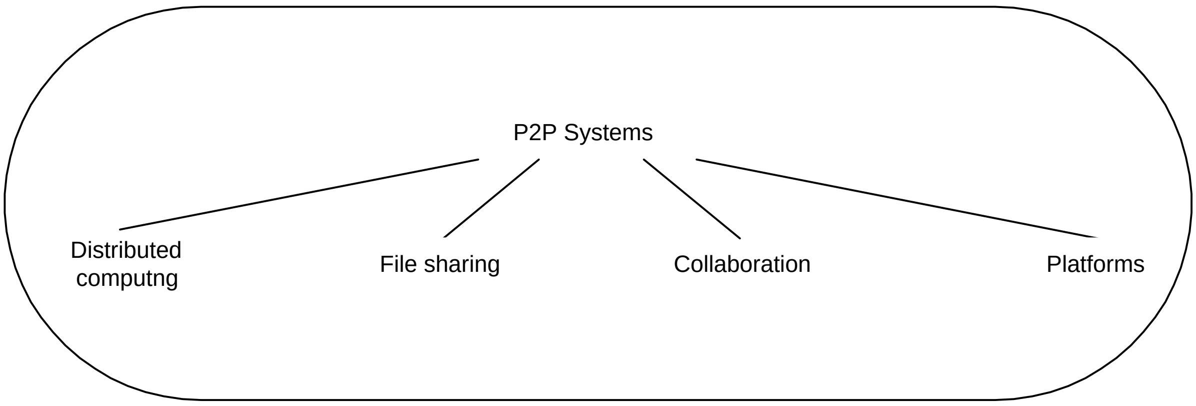


Figure 2.2: A taxonomy of P2P systems categories

enables other applications or other systems to run in a P2P fashion.

2.1.3 Pure Peer-to-Peer networks

In pure peer-to-peer systems all peers are equivalent, namely they have the same role and responsibilities. There does not exist a centralized server and clients. On the contrary, each peer is both a server and a client, it is a *servent*. All the nodes have the same responsibilities in terms of publish, download, query and communicate with any other connected node.

The Gnutella [17] belongs to the category of the pure peer-to-peer systems. The idea for this network was introduced in 2000 by tow employees of AOL's Nullsoft[1] division, Justin Frankel and Tom Pepper. Gnutella is a file sharing protocol. Users, who run software that implements the Gnutella protocol [2], are able to search for and retrieve files from other users connected to the Internet. A user must know the IP address of other Gnutella nodes in the network, in order to be connected to them. This is possible because there is a Web site where a number of Gnutella users are posed. When a user wants to find a file, he sends a query to his neighbor (the nodes that are directly connected to him). The neighbors respond, if they have results and forward the query request to their neighbors using the *flooding* protocol. The query request is forwarded for specific time, it is called time-to-live (TTL). If a search request turns up a result, the node that have the result contacts the searcher directly and then the latter is able to download the file. The Gnutella protocol does not provide a fault tolerance mechanism. In practice, searching in the Gnutella network is often slow and unreliable. Each node is a regular computer user, as such they are constantly connecting and disconnecting, so the network is never completely stable. However, various file sharing applications have been implemented using the Gnutella protocol, as for example the Limewire [23]. Other popular Gnutella clients are the gtk-gnutella

[1]http://www.nullsoft.com/

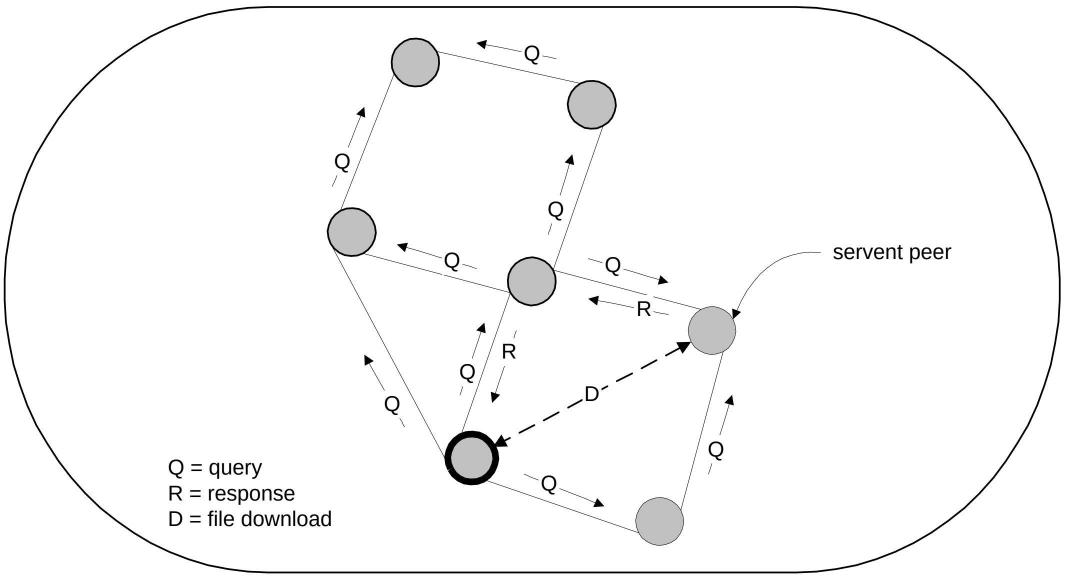


Figure 2.3: The Gnutella network

[19], the BearShare [7] and the Shareaza [42]. In Figure 2.3 is shown the Gnutella network.

Freenet [16] is another pure peer-to-peer file sharing network that initially designed by Ian Clarke [11, 12]. The main goal of Freenet is to provide an anonymous method of storing and retrieving information. A user is able to make requests for files without uncovering his identity. Also, the users store their files in the system and it is impossible to be determined who place each file to the system. The contents of each file are encrypted, and can also be broken into sections that are distributed over many different computers. Even the participants do not know what are storing.

2.1.4 Hybrid Peer-to-Peer networks

In hybrid peer-to-peer systems there is distinction between peers. The peers are not equivalent, they have different roles and responsibilities. In a hybrid peer-to-peer model there exist one or several index servers and clients that are directly connected to a server. A server obtains meta-information, such as the identity of the peers on which some information is stored. The client peers connect to a server as to publish information about the contents they offer for sharing and to search for files.

A popular hybrid peer-to-peer system is the Napster [30], it announced in January 1999 by Shawn Fanning. The Napster protocol [29] is a file sharing protocol that was aimed to share MP3 music files among Internet users. Each client peer connects to a central server and publish information about the content that it has available in its computer. The servers are organized in clusters. Each

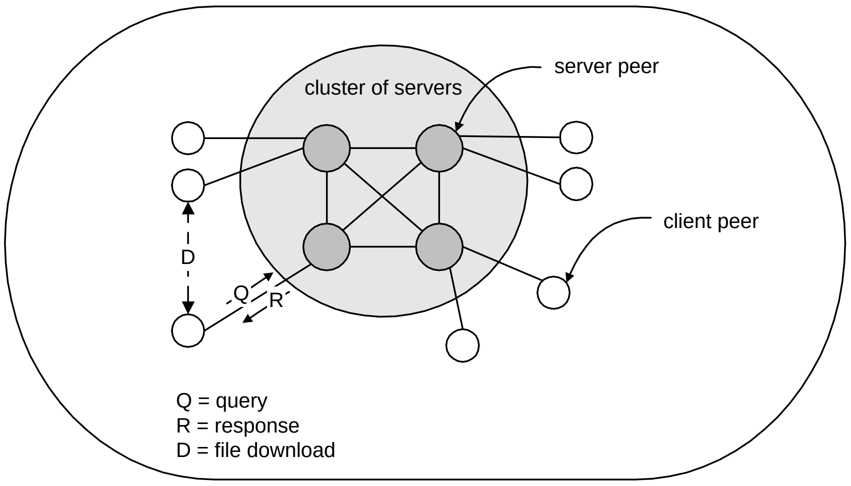


Figure 2.4: The Napster network

client peer send queries to its server, when it wants to search for a file. The servers then co-operate to process the query and return a list of matching files and their locations to the client that queries. After receiving the results, the client selects one or more files form the list and so initiates file exchanges directly from other clients. The servers also monitor the state of each peer in the system, keeping track of meta-information such as the clients' reported connection bandwidth and the duration that the client has remained connected to the network. This information is available to the client that requests for a file, so it is able to choose the best client to download a resource. The Napster network is shown in Figure 2.4.

2.1.5 Super-peer networks

As we discussed above the peer-to-peer model can either be pure or it can be hybrid. There is also an intermediate solution with the super-peers peer-to-peer model. A super-peer is a node of the network that acts as a server to a subset of clients and also it is equivalent to other peers in a network that consists only of super-peers. The query process is more efficient that the one in Gnutella, because in Gnutella all peers of the network should handle queries, unlike in super-peer networks only super-peers handle this process. Client-peers are connected to a super-peer, in a client-server way and they send to it their requests. KazaA [22] is a well-known super-peer system.

KazaA is another file shearing system that is used to exchange MP3 music files. It uses the FastTrack protocol [15]. In KazaA, users with the fastest Internet connections and the most powerful computers are automatically designated as super-peers. Peers connect to their local super-peer to upload information about the files they share and to search for files. A super-peer contains a list of some of

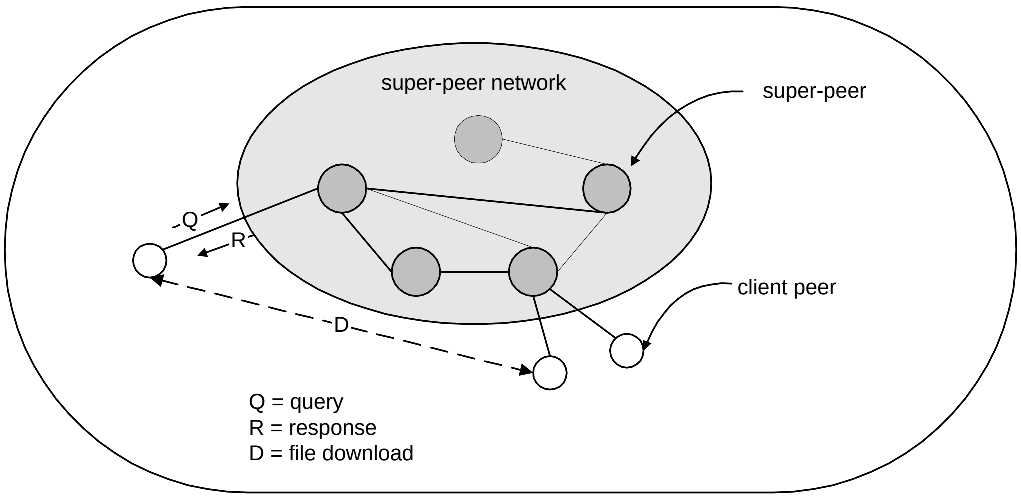


Figure 2.5: The super-peer network

the files made available by other peers and where they are located. When a peer performs a search, first searches the nearest super-peer and then the super-peer sends to the peer the results. This first super-peer refers the search to other super-peers and so on. This process is designed to make searching as fast as possible. In Figure 2.5 is shown the super-peer architecture. Another peer-to-peer resource sharing system that is based on the super-peer model is the P2P-DIET [43]. This system also provides the publish/subscribe scenarios.

2.1.6 RDF-based peer-to-peer network

Another taxonomy of peer-to-peer systems, classifies the P2P networks to these that are simple such as Napster [30] and Gnutella [17], to these that are more sophisticated, using distributed hash tables, like CAN [35] and CHORD [21], and finally to these systems that allow schema description of resources and provide complex queries using metadata, instead of simple keyword-based searches. The Edutella project [44, 31] belongs to the last category.

The Edutella network is a schema-based peer-to-peer network that use the super-peer approach. It relies on the W3C metadata RDF [48] and RDF Schema (RDFS) [9] to describe resources and is builded on the JXTA framework [4]. The Edutella network standardizes query and retrieval of RDF data, provides translation between different metadata vocabularies as to enable interoperability between different peers, defines views that join data from different metadata sources and reconciles conflicting and overlapping information. Super-peers in the Edutella network are arranged in the HyperCuP topology [37], as it is shown in Figure 2.6.

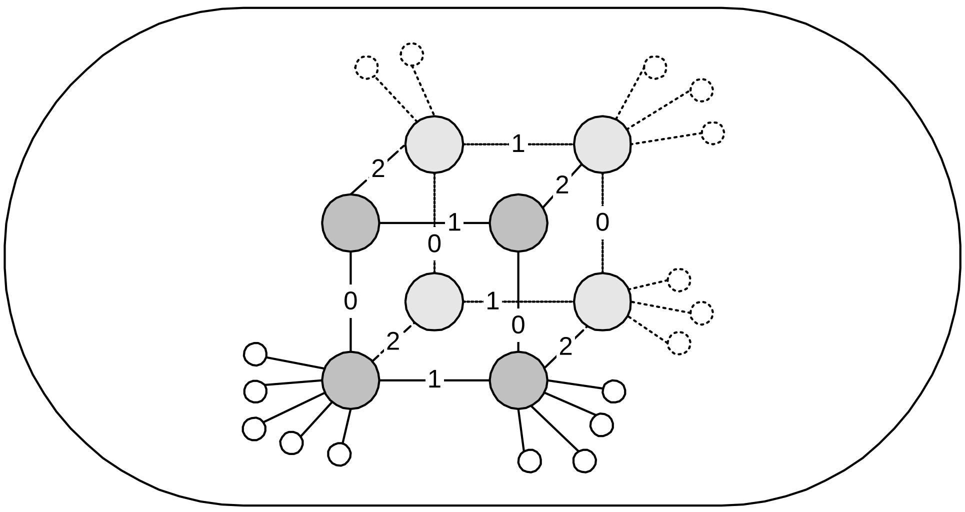


Figure 2.6: The HyperCuP super-peer topology

2.2 Background

In the following subsections we present the necessary background for this thesis. We give a brief description of the RDF data model, the Dublin Core metadata standard, the Jena API, the MySQL database and the RDQL query language.

2.2.1 The Semantic web

In our days, the amount of information that there is on the World Wide Web is huge and it is continuously increased. Most of the information is designed by humans and it is not understandable by machines, namely by the robots that browse the Web. In parallel, millions of users participate in the Web in daily basis. All these elements point to that it is difficult to efficiently search and organize the information. The requirement for evolution of the existing Web leads to the vision of the Semantic Web. The inspirator of the Semantic Web is Tim Berners-Lee[2], a software engineer at CERN[3], the European Particle Physics Laboratory.

The current Web supports documents, pages of text and figures designed for humans and understandable by humans. There exist services, i.e, search engines that search in Web pages and make directories based on the text that those Web pages contain. On the other hand, the Semantic Web vision is to make the Web machine-readable. In this way, a computer will be able to participate when inquiring and organizing information. According to the Web pages are described by metadata, namely meta-information that defines their content. Note, that the

[2]http://www.w3.org/People/Berners-Lee/
[3]http://public.web.cern.ch/public/

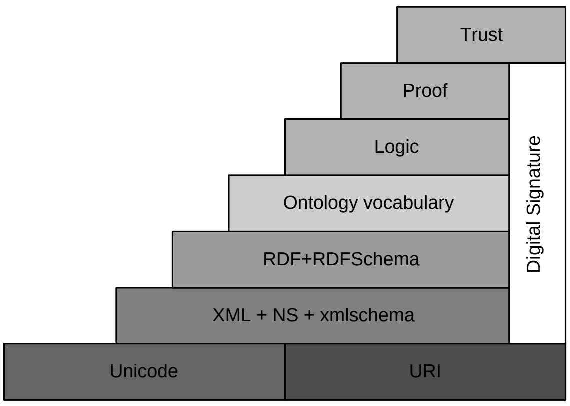


Figure 2.7: A layered approach of the Semantic Web

term metadata is not new since typical html Web pages contain metadata, for example, by using the tag “meta”. The difference with the idea of the Semantic Web vision is that metadata will describe not only information on how to read and present a Web page but will describe and organize the content of the Web pages in a structural way. The Semantic Web will have search engines too, but the difference is that the user will be able to pose more sophisticated queries than to search for a single term. For example, he will be able to ask the following question:

“Which papers include in their title the word ‘Semantic Web’ and are written by Tim Berners-Lee?”.

On the contrary, in the current Web a user has to brown selected retrieved documents to extract the information that he is actually looking for.

The component layers of the Semantic Web are shown in Figure 2.7, as have been defined by Tim Berners-Lee. At the bottom we find URIs, namely Uniform Resource Identifiers [8]. URIs are text strings that identify resources, or concepts, commonly referred to as URLs. At the next layer there is the Extensible Markup Language (XML) [13], which is an appropriate language for writing structure Web documents in a user-defined vocabulary. The XML syntax is a subset of the international text processing standard SGML (Standard Generalized Markup Language) [6] specifically intended for use on the Web. In XML there is no intended meaning associated with the nested tags, each application interpret the nesting by its way. At the next layer there is the Resource Description Framework (RDF) [25]. RDF is a language that can represent metadata as well as semantics of information in a machine understandable way. RDF builds blocks of triples (object-attribute-value), which called statements. The RDF Schema (RDFS) [9] is located at the same layer as the RDF and it provides modelling primitives for organizing Web objects into hierarchies. Actually, it is a primitive language for writing ontologies. At the higher layer the ontology languages are located, which provide the representation of more complex relationships between Web objects, than the RDF Schema can do. The logic layer enrich the ontology language. The proof layer refers to the representation of proofs in Web languages and to proof validation. At the top of the Sematic Web tower there is the trust layer. Actually, this layer represent the trust of users in the Semantic Web, namely whether the users entrust the way that the Semantic Web is organized and operates.

2.2.2 The RDF data model

The Resource Description Framework (RDF) [25] is a W3C[4] recommendation designed to describe and represent information on the Web. It is an infrastructure that enables the encoding, exchange and reuse of structured metadata, providing a means for publishing both human-readable and machine-processable vocabularies. RDF uses the XML [13] as a common syntax for the exchange and processing of metadata. RDF can be used in a variety of application areas, for example:

- in *resource discovery*, providing accurate results.
- in *cataloging*, for efficient description of Web resources.
- by *intelligent software agents*, to improve knowledge sharing and exchange.
- in *content rating*
- for describing *intellectual property rights* of Web pages.
- for expressing the *privacy preferences* of a user as well as the *privacy policies* of a Web site.

The RDF data model [48] is a way of representing RDF expressions. Two RDF expressions are equivalent if and only if they have the same data model representations. The data model consists of three object types:

- *Resources*: We consider as a resource everything that we want to describe. A resource may be a Web page, a part of it, or a collection of Web pages. In addition, a resource may be a book, an author, a paper or a computer file. Every resource has a *URI* (Universal Resource Identifier) [8]. A URI can be a URL (Universal Resource Locator) or a Web page. Note that an identifier does not necessary enable access to a resource.
- *Properties*: A property is a special kind of resource that is used as the object in an object-attribute-value triple. Actually, a property represents a characteristic of the resource to which it refers to.

[4]http://www.w3.org/

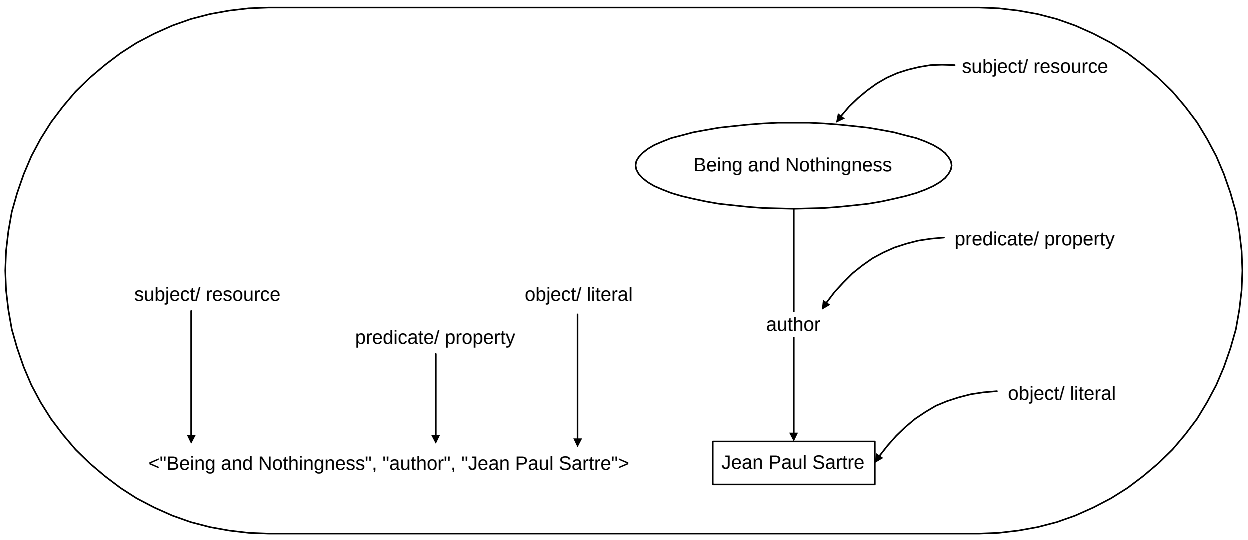


Figure 2.8: Graph representation of an RDF statement

- *Statements*: An RDF statement is an object-attribute-value triple, namely it consists of a specific resource, a named property and the relevant value of the property for this resource. The three parts of a statement are named, respectively, *subject*, *predicate* and *object*. The object of a statement can be another resource or a *literal*, namely an atomic value (string).

Consider the expression:

"Jean Paul Sartre is the author of the book *Being and Nothingness*"

This sentence has the following parts:

Subject (Resource)	Being and Nothingness
Predicate (Property)	author
Object (Value)	Jean Paul Sartre

Such expression can be represented in graph mode and as a *triple*, in the form of $subject[predicate \rightarrow object]$. In Figure 2.8 is shown the triple form and a directed graph with label nodes and arc; the arc is directed from the resource (the subject of the statement) to the value (the object of the statement).

Now consider the case that we want give some more characteristics to the resource and its author, through the following statement:

"Jean Paul Sartre born in 1905 and died in 1980, his book *Being and Nothingness* costs €16 and belongs in the area of philosophy".

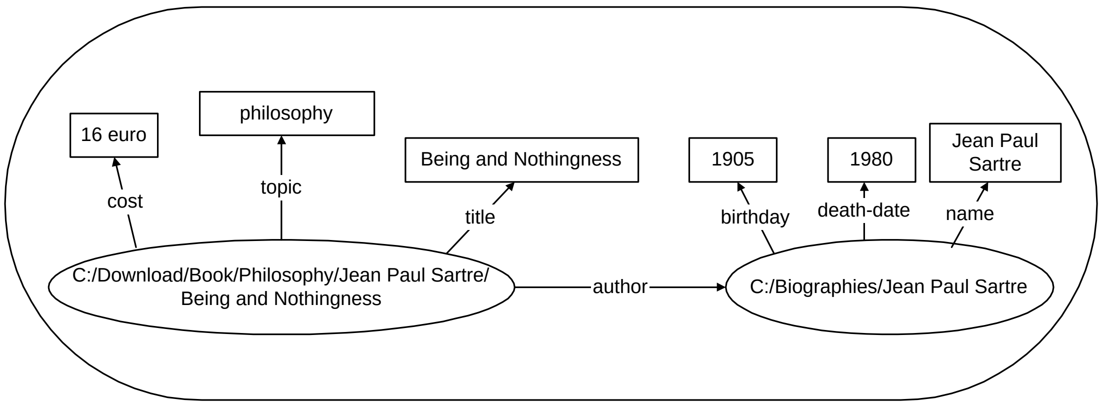


Figure 2.9: Graph representation of a complex RDF statement

The graphical representation of this statement is shown in Figure 2.9.

In order this information to be machine-processable, the RDF data model provides another representation that is based on XML syntax. The RDF document that represent the previous statement is the following one:

```
<?xml version="1.0" encoding="UTF-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns">
     <rdf:Description rdf:about="C:/Download/Book/Philosophy/Jean Paul
Sartre/ Being and Nothingness">
          <cost>€ 16</cost>
          <topic>philosophy</topic>
          <title>Being and Nothingness</title>
          <author>
               <rdf:Description rdf:about="C:/Biographies/Jean Paul
Sartre">
               <birthday>1905</birthday>
               <death-date>1980</death-date>
               <name>Jean Paul Sartre</name>
               </rdf:Description>
          </author>
     </rdf:Description>
</rdf:RDF>
```

The RDF data model defines a simple model for describing interrelationships among resources in terms of named properties and values, but does not provide mechanisms for declaring these properties, nor does it provide any mechanisms for defining the relationships between these properties and other resources. That is the role of RDF Schema (RDFS) [9]. The RDF Schema is something like a dictionary, it defines the terms that will be used in RDF statements and gives specific meanings to them. RDFS defines not only the properties of the resource (e.g., title, author, subject etc.) but may also define the kinds of resources being described (paper, Web pages, books etc.). It defines resources and properties such as `rdfs:Class` and `rdfs:subClassOf` and lets developers define a particular vocabulary for data. In other words, the RDF Schema mechanism provides a basic *type system* for RDF models.

More details on RDF and RDFS are available in [25].

2.2.3 Metadata and the Dublin Core element set

The term "meta" comes from a Greek word that denotes "alongside, with, after, next". More recent Latin and English usage employ, "meta" to denote something transcendental, or beyond the nature. Metadata can be thought of, as data about other data, descriptive information about Web resources. Within the context of the World Wide Web metadata may be used for information discovery, but is also important in the context of cataloguing resources.

The Dublin Core metadata standard is a simple attribute set for describing a wide range of Web resources. The Dublin Core Metadata Element Set [3] has been endorsed by the Dublin Core Metadata Initiative[5]. It has been kept as small and simple as possible, in order to allow the simple description for information resources, while providing effective retrieval of those resources in the Web. The Dublin Core can help the "digital tourist" find the object of his research, by supporting a common set of elements that are semantically intelligible by universe. The Dublin Core standard includes two levels: the *Simple* and the *Qualified*. The Simple Dublin Core consists of fifteen elements, while the Qualified Dublin Core employs additional qualifiers to further refine the meaning of a resource.

In our system, we chose to use the Simple Dublin Core for the description of our resources. The fifteen elements are: title, creator, subject, description, publisher, contributor, date, type, format, identifier, source, language, relation, coverage and rights. Detailed definitions of these elements are given on the Dublin Core Web Site [1]. All Dublin Core elements are optional and repeatable.

A graphic representation of the Dublin Core element set is shown in Figure 2.10. The described resource is denoted as an ellipse, while the value that each element of the Dublin Core has is denoted as a box.

2.2.4 The Jena API

Jena[6] is a research project that has been implemented by Hewlett Packard research Labs[7]. Actually, Jena is a collection of RDF tools written in the Java programming language[8] that includes:

[5]http://dublincore.org/
[6]http://www.hpl.hp.com/semweb/jena.htm
[7]http://hpl.hp.com
[8]http://java.sun.com

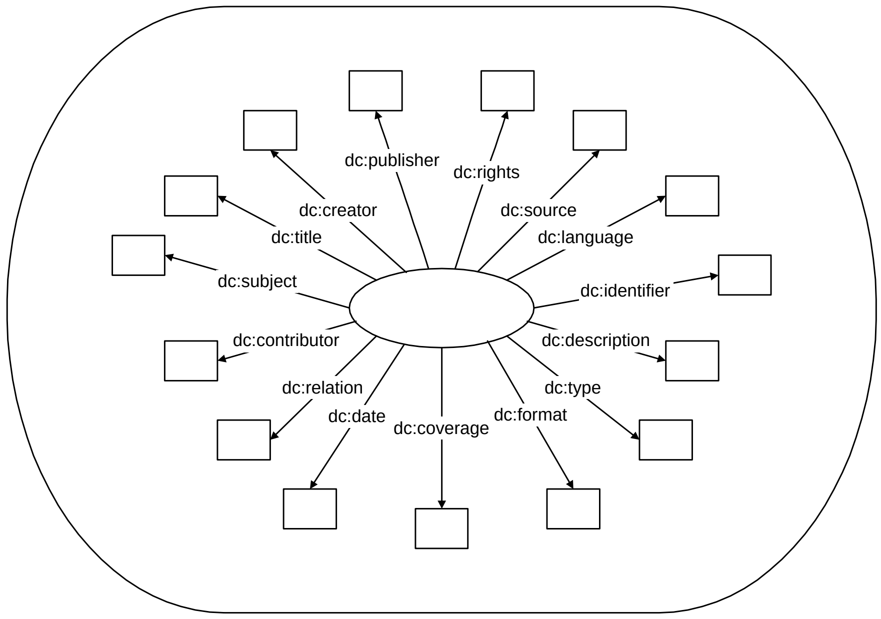


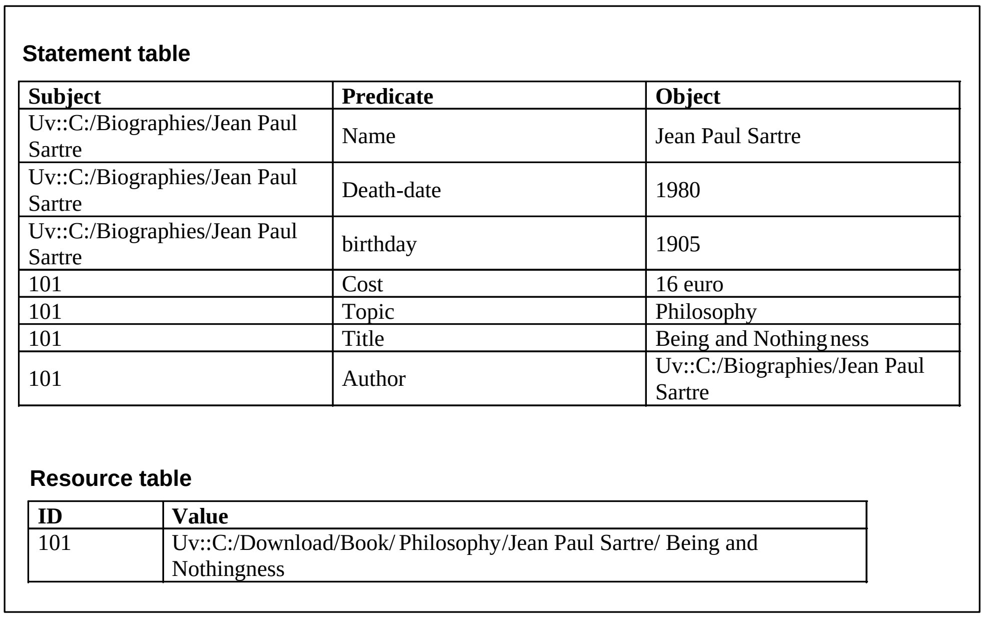
Figure 2.10: The Dublin Core element set

- A Java API.
- ARP: An RDF parser.
- RDQL: A query language.
- Support classes for DAML+OIL ontologies.
- Persistent storage based on various relational databases.

The goal is to speed up the design and implementation of Semantic Web applications. This Java toolkit can be used for the efficient creation and manipulation of RDF graphs like the one shown in Figure 2.8.

In the Jena API [26, 40, 10, 47] an RDF graph is called *model*. An RDF graph contains one or more RDF statements which in the Jena API are called *statements*. The Jena API uses object classes to represent graphs, resources, properties and literals. In addition, special methods have been implemented in order to read and write RDF files. This toolkit uses a database engine in order to store the statements that constitute a model.

The Jena2 API is the second generation RDF toolkit that follows the Jena1 API. It can manage more than one model in a single database by storing each model in a separate table. Moreover, the Jena2 API uses a denormalized schema, in which the resources, the URIs and the simple literal values are stored directly in the *statement table*. In the case that there is need to store long literals or URIs, namely the length of these is bigger than a threshold, then the Jena2 API



Statement table

Subject	Predicate	Object
Uv::C:/Biographies/Jean Paul Sartre	Name	Jean Paul Sartre
Uv::C:/Biographies/Jean Paul Sartre	Death-date	1980
Uv::C:/Biographies/Jean Paul Sartre	birthday	1905
101	Cost	16 euro
101	Topic	Philosophy
101	Title	Being and Nothing ness
101	Author	Uv::C:/Biographies/Jean Paul Sartre

Resource table

ID	Value
101	Uv::C:/Download/Book/ Philosophy/Jean Paul Sartre/ Being and Nothingness

Table 2.1: RDF information stored in a MySQL table

creates separate tables for the resources and literals. Note, that this is the only case that separate tables are created for resources and literals. In this way, the Jena2 API avoids a big number of joins when retrieving statements which comes with a cost in storage space due to the separate tables. The Jena2 toolkit stores the RDF triples in a remote database accessed via JDBC.

Consider the case that the information of the RDF graph in Figure 2.9 is stored in a database. Then, the RDF statements will be stored in the statement table, while a separate *resource table* will be used, as the long URI "$C : /Download/Book/Philosophy/JeanPaulSartre/BeingandNothingness$" to be stored in it. The tables are presented in Table 2.1.

Moreover, the Jena2 API supports the RDQL [27, 10, 39] query language. RDQL is a query language for RDF data. By writing an RDQL query, we can retrieve answers from a database, in which RDF data are stored. The RDQL query language is discussed in more detail in Section 2.2.6. The structure of the Jena2 API implementation is shown in Figure 2.11.

The Jena API supports three different database engines: (a) MySQL[9], (b) Oracle[10] and (c) PostgreSQL[11] in two platforms, Linux and WindowsXP. We have implemented our system using the Jena2 API and the MySQL database in a WindowsXP System.

[9]www.mysql.com
[10]www.oracle.com
[11]www.postgresql.org

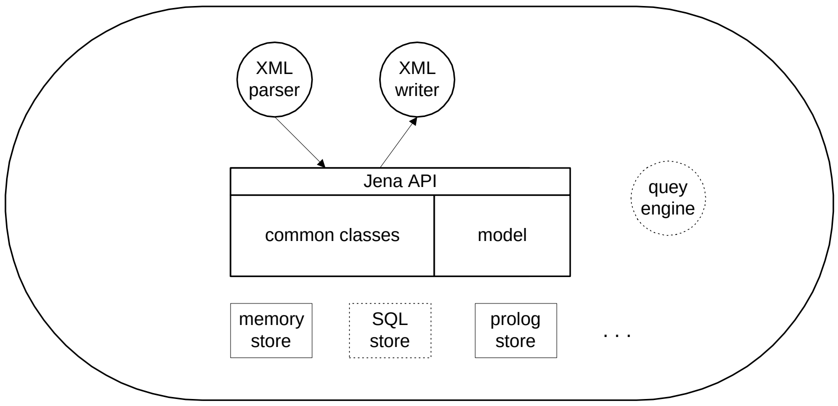


Figure 2.11: The architecture of the Jena2 API

2.2.5 The MySQL database

The MySQL database [14] is an open-source SQL[12] database system, which is developed and supported by the commercial company MYSQL AB[13]. It is a relational database management system, which means that it is a structured collection of data that are stored in separate tables and not in one big storeroom.

The Jena2 API has been tested with the MySQL Standard binary release. The Jena tables are created using InnoDB tables which provide ACID transaction support (atomicity, concurrency, isolation, durability). Any modifications to the Jena database models are immediately written to the database. If the application has an open transaction, the changes are committed when the application does a commit. The Jena API communicates with the MySQL database through a JDBC driver, in order to store and to manipulate RDF information.

2.2.6 The RDQL query language

RDQL[27, 39] stands for RDF Data Query Language. It is a query language for RDF [25] data in Jena models. The idea is to provide a data-oriented query model so that there is a more declarative approach to complement the fine-grained, procedural Jena API. It is "data-oriented", in the sense that it only queries the information help in the models, there is no inference being done. Of course, the Jena model may be "smart" in that it provides the impression that certain triples exists by creating them on-demand. However, the RDQL system does not do anything other than take a description of what the application wants, in the form of a query, and return that information, in the form of a set of bindings. RDQL is an implementation of the SquishQL [27]. SquishQL is an SQL-like query language that matches a graph pattern to a data source, derived from rdfDB [20]. This

[12]http://www.microsoft.com/sql/
[13]http://www.mysql.com/company/

class of query languages regards RDF as triple data. An example RDQL query is the following:

```
SELECT ?x
WHERE (?x, <topic>, "philosophy"),
      (?x, <author>, ?y)
      (?y, <name>, "Jean Paul Sartre")
```

In this query, we want to find a node in an RDF graph, ?x, which has the property *topic* with the value "philosophy" and the property *author* has as *name* the value "Jean Paul Sartre".

Considering the RDF graph of Figure 2.9, when we execute the above query we will get the answer:

```
x
=======================
<http://book/Being and Nothingness>
```

2.3 Summary

In this chapter we presented the necessary background for the understanding of this thesis. We presented the basic characteristics of the Semantic Web and we discussed about the tools that we used for the implementation of our semantic application. In the next chapter, we discuss about the architecture of our system and its basic functionalities.

Chapter 3

System Architecture

In the previous chapter we discussed some alternative peer-to-peer architectures and some well-known peer-to-peer systems. We also gave the necessary background for this thesis. In this chapter, we study the architecture of our system and how the desired functionalities can efficiently be supported using such an architecture. Our goal is to build a hybrid peer-to-peer resource-sharing system, which provides semantic interoperability among data sources, so as to execute queries. Our current implementation is a file-sharing application and unless stated otherwise we will consider such applications as our main example scenario.

3.1 Architecture

We have designed and implemented a *hybrid peer-to-peer* resource sharing system. There are two kinds of nodes in our system; the *server-peers* and the *client-peers*. A single network of the system consists of one server-peer and a large number of client-peers. A high level view of the system is shown in Figure 3.1.

A server-peer can be seen as the coordinator for the rest of the peers in its network. A server-peer handles all queries of the client-peers that are directly connected to it and manages information on the schema or the metadata that each one of those client-peers supports. Each client-peer is attached to the network through a single server-peer. This server-peer is the *access point* of the client-peer to the network. A client-peer sends all its requests to its access point, but when it comes to request an actual resource, the client-peer requests the resource in a pure peer-to-peer way from the client-peer which owns it and not through a server-peer. We distinguish between two types of client-peers; the *normal* ones and the *volunteer* ones. A normal client-peer uses the system in such a way that it sacrifices the less possible amount of its system resource, namely cpu cycles and bandwidth. On the contrary, a volunteer client-peer tries to “take” load from the server-peer, for example, by answering a fraction of the queries posed to the server-peer. In addition, an important system characteristic

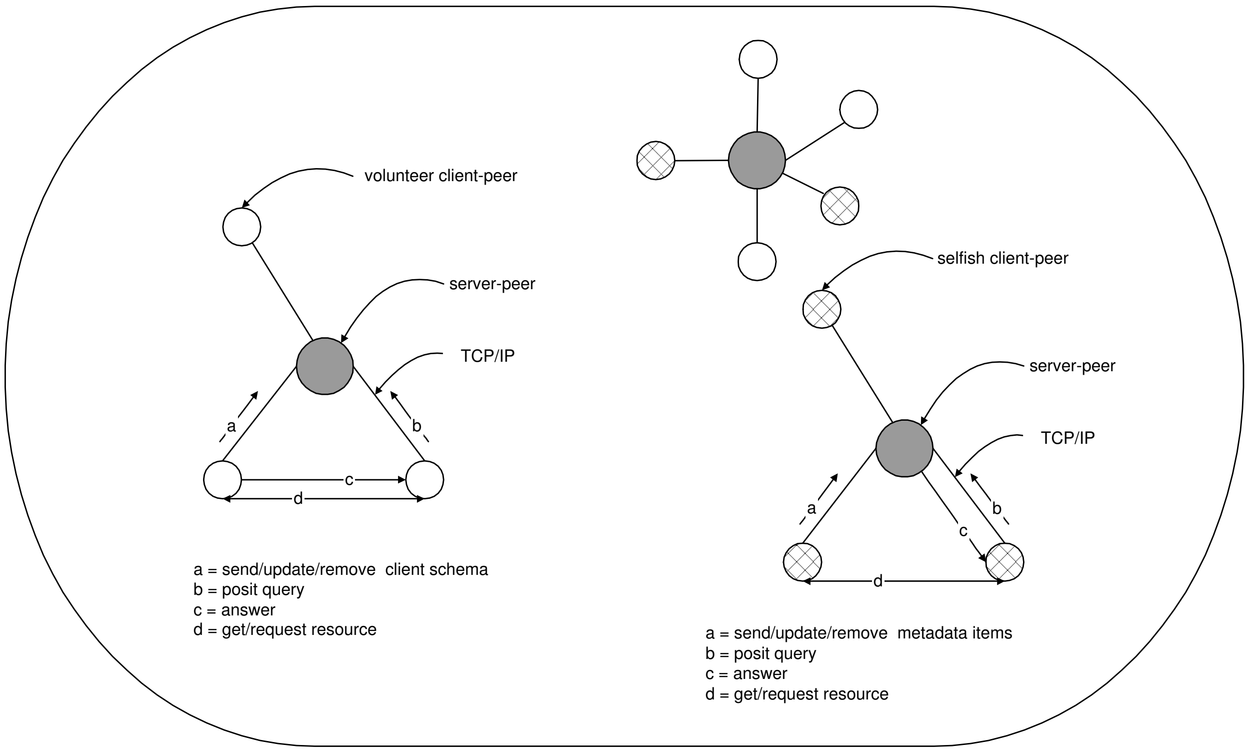


Figure 3.1: System architecture

is that we allow client-peers to use their own defined schema in order to describe their resources. This functionality allows client-peers, which already have their resources described in a schema different than the one of the server-peer, to participate in the system in a transparent way. At last, periodic fault-tolerant operations of the server-peers detect and handle any possible client-peer failures or silent-disconnections. In the current implementation, resources are files that a client-peer stores locally and wants to share with other client-peers of the system. The data model used to describe resources is the RDF data model [25], while the RDQL query language [39] is used to define queries. Finally, a MySQL database is used locally at each peer to store RDF data.

3.1.1 System functionalities

Let us now give a brief description of the functionalities that our system supports. A client-peer may *publish* a resource, so other client-peers may see it and request it. A published resource is described by metadata that a client-peer defines. In our file-sharing scenario, a resource can be a file of any type, for example, it can be a music file, an image file or a document file. A resource can also be *removed* or *updated* (update the metadata that describe it) by the client-peer that originally has published it. In addition, our system supports the *query scenario*, namely a client can pose a query to the system in order to find matching resources owned by other client-peers. The system will immediately reply with pointers to matching resources owned by client-peers that are on-line at the moment.

3.1.2 Normal and volunteer client-peers

Client-peers usually have different characteristics with respect their capabilities; particularly in terms of available bandwidth, storage space, precessing power and memory. In order to build an efficient peer-to-peer system, once should take into account the characteristics of participating peers, including their heterogeneity [24, 36].

We distinguish between two types of client-peers; the normal ones and the volunteer ones. A normal client-peer uses the functionalities of the system by sacrificing the less possible amount of its own system resources, namely cpu cycles and bandwidth. On the other hand, a volunteer client-peer can offer a fraction of its system resources in order to remove load from the server-peer. Automatic recognition of client-peers, good enough (powerful or idle) to offer their resources or policies of giving awards (certain privileges) in order to lure client-peers and become volunteers are out of the scope of this thesis. Current work on these area includes [45, 33]. The main goal in these articles is the identification of peers as *altruistic peers* and *selfish peers*. In our system, we assume a cooperative environment where a number of client-peers will be "nice" enough to be a volunteer when it is possible (idle pc or powerful enough).

Let us now give a high level view of the differences between normal and volunteer client-peers in our system. Both types of client-peers, use the same functionalities of the system. The main difference is the way that each client-peer and its access point execute these functionalities. A normal client-peer does not only store locally the metadata items of its resources, instead it sends those metadata items to its access point. Thus, it has to communicate with its access point each time it publishes, removes or updates a resource leading to higher communication cost for the network and computational cost for the server-peers. More importantly, the access point has to answer queries matching these resources (search for the matching resource and reply with an answer), instead of just forwarding queries to client-peers with a matching schema. On the other hand, a volunteer client-peer follows a schema based rooting strategy. A volunteer client-peer does not send the metadata items of its resources to its access point. It sends only the schema that it supports and when the access point receives queries that match this schema, the volunteer client-peer will answer them. A volunteer client-peer does not communicate with its access point every time it publishes, removes or updates a resource, instead it communicates only when its schema is updated. Another RDF-based P2P network that follows a similar strategy, is the Edutella system [32, 46]. In addition, in our system a volunteer client-peer that uses its own defined schema, makes the translations on its own, whether the access point makes all necessary translations for a normal client-peer. All these will become more clear when reading one by one the functionalities in Chapter 4.

The distinction between normal and volunteer client-peers serves some application scenarios. We decided to design our system in such a way, because in reality the devices that participate in the network have alternative characteristics that may determine their action. It is advisable that a client-peer becomes a normal peer, in the case that it can't afford a high powered computer; a client-peer that choices to become a normal peer, it sacrifices the smallest amount of system resources that are demanded. For example, a normal client-peer does not receive queries from its access point, so does not search in the local database for matching resources, neither translates incoming or outgoing messages. Thus, a normal client-peer could be a low powered computer. Another case, in which a client-peer selects to become normal peer, is when it runs a lot of applications simultaneously, so it prefers to participate in the network as the light type of client-peers. In the case that a client-peer is connected to the network through a low-speed connection, it is better to become a normal client-peer than a volunteer. A volunteer client-peer receives queries from its access point, searches in its local database as to find matching resources and then, if it creates answers, it connects to the requestor client-peer. So in the case that the system consists of volunteer client-peers that have slow Internet connection speed, then a general delay in the performance of the system, will take place. Therefore, it would be better peers that have slow connections to become normal client-peers. In addition, a client-peer that does not have a large number of available resources and it is just interested to find than to share resources, is better to become a normal client-peer.

On the other hand, a volunteer client-peer should be a peer that participates in the network using a powerful computer, because the role of a volunteer is by comparison more demanding in cpu cycles and bandwidth than the role of a normal client-peer. For example, an idle computer can be a volunteer client-peer because it can offer all its system resources. Similarly, a client-peer that has a large number of available resources, is better to become a volunteer than a normal client-peer. If it becomes normal, it weights the server-peer down, because it demands storage space of the database of the server-peer for the metadata of its available resources, and also provokes the server-peer to search by itself in a huge database, which means space and time cost for the server-peer. Another case in which a client-peer is better to become a volunteer one, is if it is connected to the network through a high-speed connection. Finally, it would be a good idea if there were policies of given awards for the volunteer peers, as for example the queries of these peers to be forwarded by the system faster, than the queries of the normal client-peers.

To sum up, in an efficient application scenario, normal client-peers is better to become peers that are low-powered, having small number of available resources

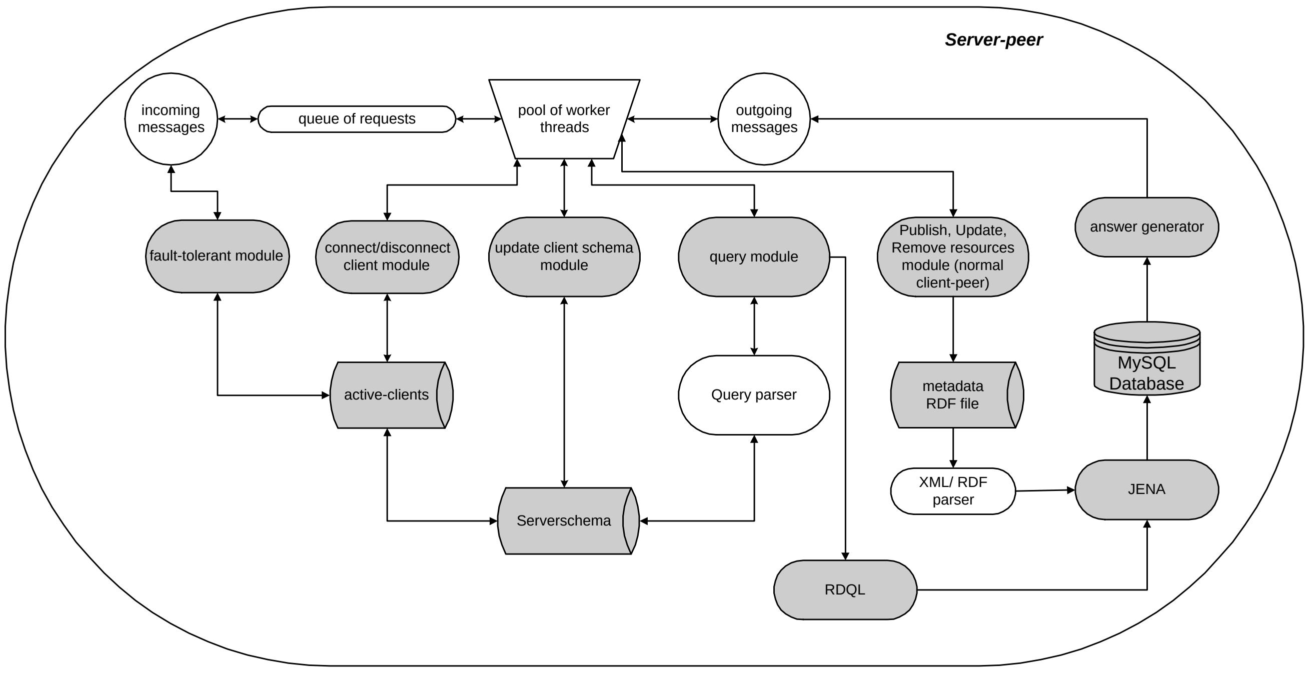


Figure 3.2: The server-peer architecture

and slow Internet connection, that wants to search and obtain resources. These can be small devices like mobile phones, personal home computers, laptops and hand-held computers. On the contrary, volunteer client-peers is better to become idle and powerful computers, that have a large number of available resources and high speed Internet connection. These type of devices are computers that are located for example in a library or in a university and work on information exchange. In this thesis we have not implemented an automatic recognition of which client-peers should be normal or volunteer.

3.2 The server-peer

In the previous section we gave a general description of the architecture of our system, while in the following ones we discuss in more detail the separate components of our design. In this section we discuss about the internal architecture of server-peers and their reactions to requests from client-peers. A server-peer is one of the two kinds of nodes, which constitute our system. A server-peer works continuously and autonomously to serve the connected to it client-peers. The internal architecture of a server-peer is discussed in the subsections below and it is shown in Figure 3.2.

3.2.1 Data structures

The role of a server-peer is to handle and to serve efficiently every request that receives from the client-peers that are connected to it. A server-peer can be thought as a coordinator. In order to achieve the correct and efficient execution of requests, each server-peer manages two data structures, one to store information on online client-peers and one to store the *schema* that it supports, called *active-clients* and *serverSchema* respectively. By the term schema we define the set of attributes that are used for the description of resources[1]. A view of these data structures is shown in Figure 3.3.

The data structure active-clients is a *java hash table*. Each *record* of this hash table contains information on one of the online client-peers. The IP address of a client-peer is used as input to the hash function in order to create the key of each record. In addition, each record contains the string values of the IP address and port of a client-peer. At last, each record contains a list of pointers to objects (records) of the serverSchema data structure. The serverSchema data structure is also a java hash table. Each record of this hash table contains information on one of the attributes of the schema that the server-peer supports. This time, the name of the attribute is the input of the hash function so as to create the key of each record. The values that are contained in each record, are the name of the attribute and a URL representation of it. At this point we can clarify the purpose of the list of pointers in the records of the active-clients data structure. Each record r in the active-clients data structure, represents a client-peer c. Each pointer p of the list of pointers in r, points to a record r' in the serverSchema data structure. The record r' represents an attribute a. Thus, the pointer p "means" that c *supports* the attribute a.

3.2.2 Handling requests

A server-peer uses standard multithreading techniques in order to handle incoming requests. More precisely, the *thread-pool* [38] model is used. Upon start-up time, a server-peer creates a pool of *worker threads*. The number of worker threads is a system parameter. When a request arrives, the server-peer assigns the request to one of the threads in the pool. A worker thread returns in the pool after successfully handling a request. There are four kinds of requests that a server-peer handles:

1. the *connection/disconnection* request.

2. the *update schema* request.

3. the *update metadata item* request.

[1]Detailed discussion on the schema of peers in our system, can be found in Chapter 4

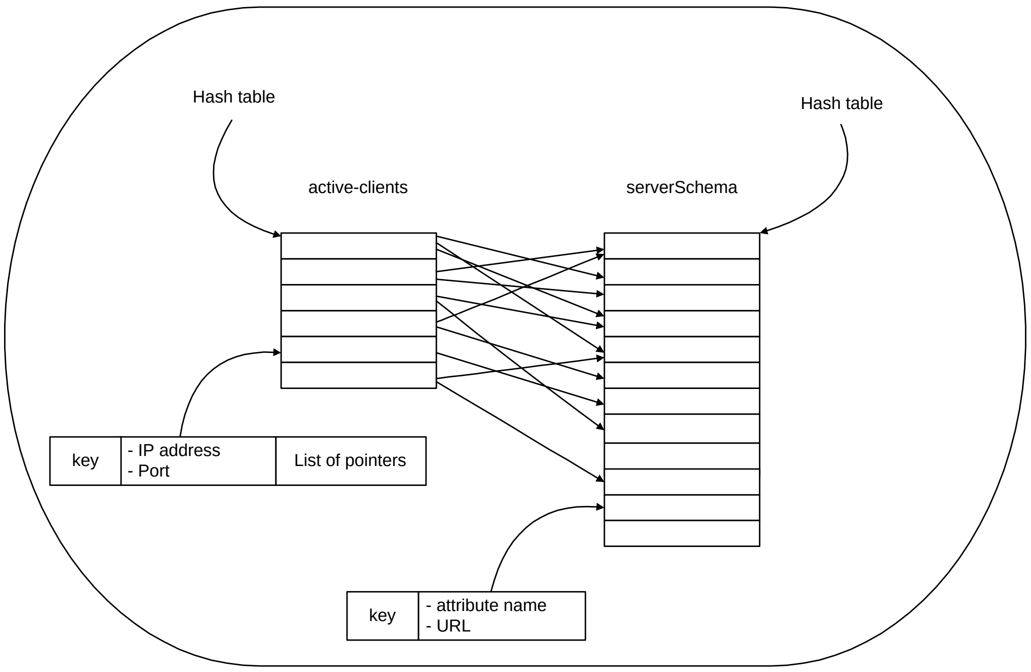


Figure 3.3: The active-clients data structure and the serverSchema data structure

4. the *query* request.

In the following subsections we describe the way that a server-peer handles the above requests. Note, that for readability reasons we omit the discussion about the query request, which we describe in detail in Chapter 4.

3.2.3 Handling the connect and disconnect request

A connection or disconnection request can be sent by any client-peer (volunteer or normal) to any server-peer in the system. A client-peer sends a connection request when it wants to connect (or reconnect) to the network. As it is obvious, the connection request is the only type of request that a client-peer can send without already being online. In the case that the client-peer is a volunteer one, the connection request contains (a) the IP address and the port of the client-peer and (b) the schema that the client-peer supports. In the case that the client-peer is a normal one, the connection request contains (a) the IP address and the port of the client-peer, (b) the metadata item of the resources that the client-peer had published, during previous online sessions (if any) and (c) in the case that the client-peer has defined its own schema, the relations between this schema and the schema that is supported by the server-peer. The latter stands only in the case that the client-peer was previously connected to the same supper-peer (reconnection), which means that it has already created the relations.

When a server-peer x receives a connection request from a client-peer c, it has to update the local active-clients data structure. Thus, it creates a record r to contain the IP address and the port of the new on-line client-peer c. Then, in the case that c is a volunteer client-peer, x initializes the list of pointers in r by creating one pointer for each attribute in the schema of c that points to the appropriate attribute record in the serverSchema data structure (easily found by hashing the attribute name). In the case that c is a normal client-peer, x puts all the metadata items of c in the local MySQL database and stores the relations between the schema that c supports and its schema.

In reverse, a client-peer can send a disconnection request in order to disconnect from the network. Of course, both volunteers and normal client-peers can send disconnection requests. A disconnection request just contains the IP address of the client-peer. When a server-peer x receives a disconnection request from a client-peer c, it has to update the active-clients data structure. Thus, x removes the record that represents c from the active-clients data structure. In the case that c is a normal client-peer, all the metadata items of the resources that c owns, are removed from the local MySQL database and the relations between its schema and the schema of the server-peer, are removed too.

3.2.4 Handling the update schema request

The second kind of request that a server-peer may receive is an update schema request. Only volunteer client-peers may send this type of request. An update schema request contains (a) the IP address and the port of the client-peer and (b) the schema that the client-peer supports. Assume, an on-line client-peer c sends to its access point x an update schema request. The server-peer x, has to update the local active-clients data structure. It tracks down the record r that represents client-peer c (by hashing the IP address of c) and updates the pointer list in r, so as to point only to attributes that are contained in the schema of c. Note, that a client-peer c sends an update schema request, only if c has just removed/updated a previous published resource or if it has just published a new one. In both cases, the schema that c supports may change and if it does, then c informs its access point on that.

3.2.5 Handling the update metadata item request

The third kind of request that a server-peer may receive is an update metadata item request. The client-peers that are able to send this request are the normal client-peers. An update metadata item request contains (a) the IP address and the port of the client-peer and (b) in the case that the client-peer wants to publish a new resource or to update an already published resource, the request contains the metadata item of this resource, in the case that the client-peer wants to remove a published one, the request contains just the name of the resource. For example, an on-line normal client-peer c sends to its access point x an update metadata item request, when publishing a new resource. The server-peer x inserts the metadata item of the resource in the local MySQL database. In the case that c wants to update the metadata item of a published resource, the server-peer x deletes from the database the previous metadata item that described the specific resource and then inserts the new metadata item. Finally, if c wants to remove a published resource, then x deletes the metadata item of the resource.

3.2.6 Fault-tolerance

A client-peer may fail and become unreachable. For example a crash on the system of a client-peer or a silent disconnection (the user turns off the application without pushing the disconnection button) may be the reason for a client-peer to fail and bring a number of problems to the system. Consider the case that a server-peer wants to forward a query to a client-peer that is supposed to be on-line, but the client-peer is off-line. If this happens for a large number of client-peers, then a large number of resources of the server-peer are waisted trying to communicate with those client-peers. Thus, in order to avoid such problems each server-peer manages the data structure active-clients, where it stores all client-peers which are online and connected to it. A server-peer updates this structure each time a new client-peer requests to be connected to or an already connected client-peer requests to be disconnected from the system. We have also implemented a mechanism which is responsible to periodically check all the on-line client-peers that are connected to a server-peer and update the data structure active-clients if needed. This happens with period T, which is a system parameter. For our example we used $T = 5$ minutes. A server-peer x checks all client-peers with a record in the local active-clients data structure in the following way:

1. x sends a *ping* message to all the client-peers in the active-clients data structure.

2. Then x waits for time T_1 and stores all replies.

3. Each client-peer that receives a *ping* message, replies with a *pong* message to x (the sender of *ping* message).

4. After time T_1, x checks if all client-peers have replied. For all the client-peers that have not replied, x removes them from its active-clients data structure and deletes all information about their schema.

Time T_1 is a system parameter. For our examples we used $T_1 = 3$ minutes.

3.3 The client-peer

A client-peer is the second kind of node that constitute our system. As we have already discussed, a client-peer is a computer of a real user that can share resources or pose queries to the system in order to locate and acquire interesting resources. A high level view of the internal architecture of a client-peer is shown in Figure 3.4.

3.3.1 Functionalities of client-peers

The client-peer application offers a number functionalities to its user. The GUI mediates between the user and the client-peer, so as to forward the demands of the former to the latter and messages from the client-peer to the user. The functionalities that a client-peer offers are:

1. *connection/disconnection* from a server-peer
2. *publish* a resource
3. *remove* a resource
4. *update* a resource
5. *query* for a resource

The first functionality, *connection/disconnection* from a server-peer, refers to the opportunity that a client-peer has, to select the access point to which it wants to be connected. While a client-peer is connected to a server-peer, can participate in the system and is able to use the rest of the functionalities. When it decides to terminates the connection to its access point, it selects the functionality of disconnection. Latter the client-peer can reconnect to the same server-peer or connect to a different one.

The second functionality that a client-peer has, is the functionality of *publishing* a resource. Using this functionality, a client-peer can share its resources with the other client-peers of the network. A client-peer c that publishes a resource r, creates a *metadata item* for r, namely data that describes r. This metadata item, denoted by $metadata(r)$, is a set of *attribute value* pairs. The metadata item of a published resource is stored locally or forwarded to its access point. This depends on weather the client-peer is normal or volunteer. The actual resource is stored, at both cases, locally at the client-peer that published it.

A client-peer can also *remove* a resource that it has already published. This functionality is useful in the case that a user of a client-peer application no longer wants to share a resource. In addition, a client-peer can *update* the metadata item of a resource that was previously published by it. The user is able to change the values of the attributes in the metadata item of a resource. He is also able to

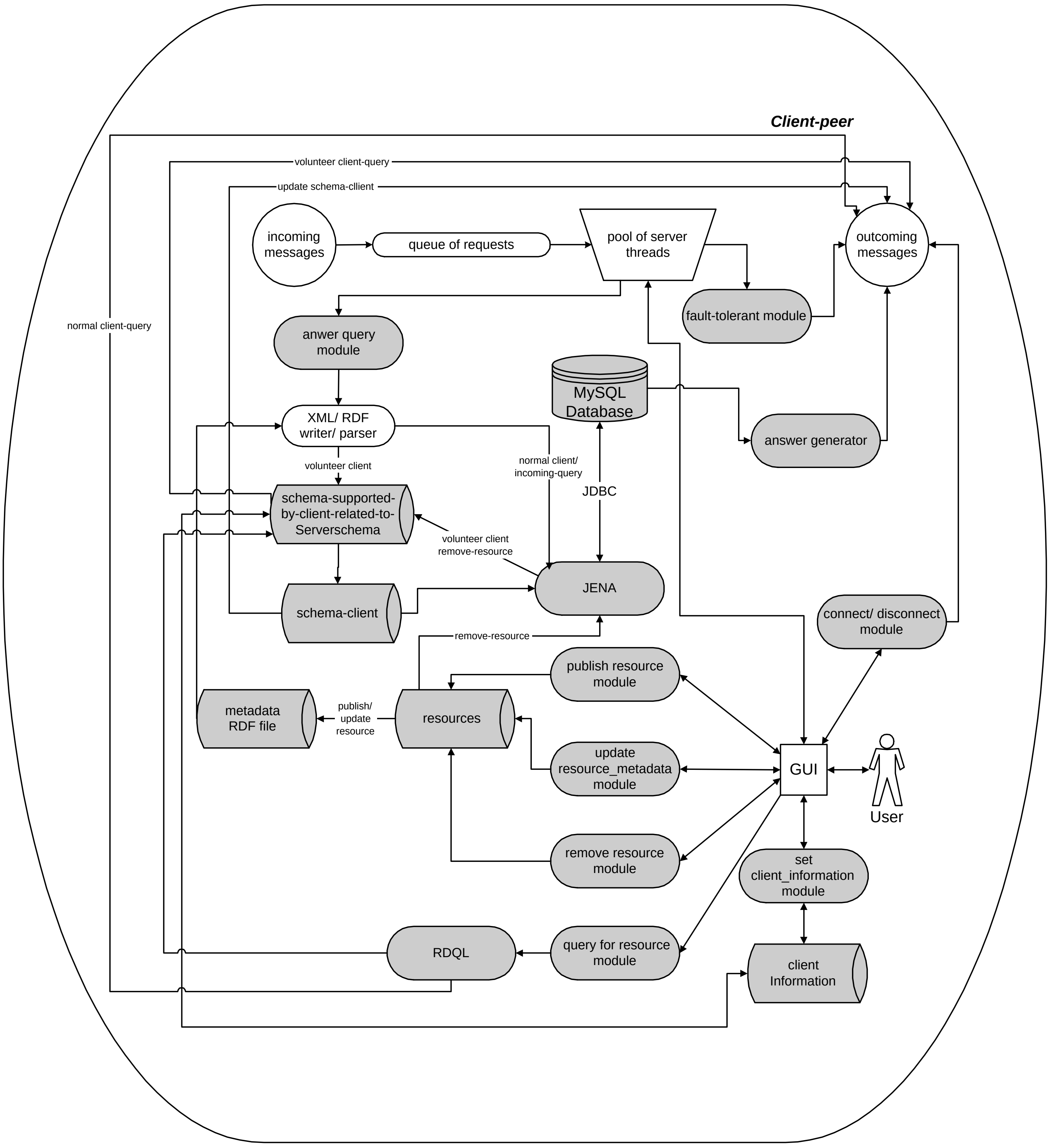


Figure 3.4: The client-peer architecture

add new attribute value pairs, in order the description of the resource to become more explicit, or he can delete attribute value pairs that considers that are untrue or do not respond to the resource any more.

Finally, the client-peer has the capability to pose queries to the system in order to find interesting resources. A query describes the “features” that the matching resources should have. A query is written in the RDQL query language [39], which means that the users of the client-peer application should be able to understand and write in the RDQL query language (expert users). The system will immediately reply with “pointers” to matching resources. This means that an answer does not contain an actual matching resource but a pointer to the location (IP address, port and name) of the resource. Then, the client-peer that posed the query, can connect directly to the client-peer that is the owner of a matching resource and request the resource.

3.3.2 Set client-peer information

When a user starts the client-peer application program for the first time, he defines a set of information. He specifies the port in which the application is going to listen for messages. He, also, defines the server-peer, to which he wants to be connected, and the database, in which the metadata items of his published resources are going to be stored. Moreover, the user defines the schema that intends to use for the description of the selected to be published resources. The client-peer application program uses XML files to store the program parameters. Thus, all information is stored even if the user terminates the application or turns off his computer. All the above information can be updated at any time.

3.4 Summary

In this chapter we presented the architecture of our system and the basic modules of the server-peers and client-peers. Moreover, we included a high level description of the functionalities that our system supports. In the following chapter, we discuss in detail the functionalities of the system with emphasis on the schema-based routing strategy.

Chapter 4

Schema-based Routing

In the previous chapter we presented the architecture of our system. We discussed about the internal architecture of the partial components that constitute the system, namely the server-peers and the client-peers. In this chapter, we discuss various issues regarding schemas that different types of peers use and then we analyze in more detail the network functionalities of the system and how these functionalities are executed in a schema-based routing way.

4.1 Schema

As "schema" is defined the mechanism that declares the properties of a resource and the constraints on relationships between these properties and other resources. A schema defines not only the properties of a resource (e.g., title, author, subject, size, color, etc.) but may also define the kinds of resources being described (books, Web pages, people, companies, etc.). It defines the data type of resources and properties. We consider that our application scenarios, intend to be used as a simple file sharing application, in which users share simple described resources. Thus, by the term schema we just mean the metadata vocabulary, namely the attribute set that can be used for the description of a resource.

Each peer in our system has a metadata vocabulary. In the case that it is a server-peer, it uses the metadata vocabulary in order to communicate with its client-peers. A client-peer has the capability to accept the vocabulary of a server-peer or to define its own vocabulary. A client-peer uses its vocabulary in order to describe its available resources and to query the system for desirable files. We say that a peer *supports a schema*, when this peer understands the semantically meaning of the attributes that constitute this schema. Then, this peer is able to communicate with other peers using this attribute set. In the following two subsections we describe the schema and its usage in both server-peers and client-peers.

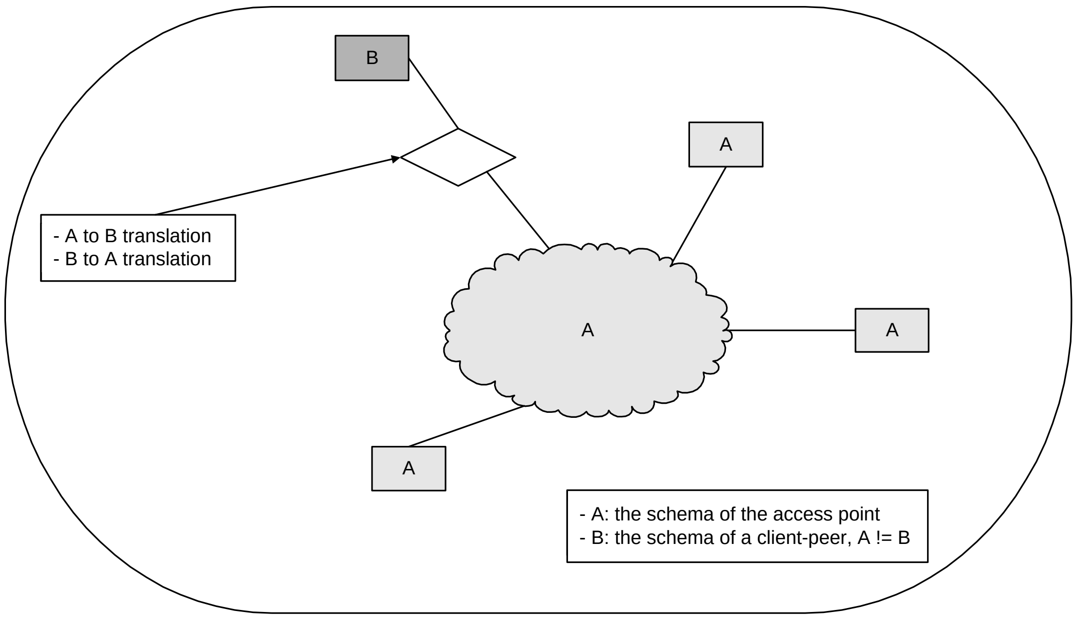

B
A
- A to B translation
- B to A translation
A
A
A
A
- A: the schema of the access point
- B: the schema of a client-peer, A != B

Figure 4.1: Schema translation

4.1.1 The schema supported by server-peers

Each server-peer in the network supports a schema. The schema that a server-peer x supports, is the return value of the function $serverSchema(x)$. The usage of the schema of a server-peer can be better understood along with the role of the server-peer as a coordinator. The schema that a server-peer supports can be seen as a *global* or *middleware* language for the network. The client-peers use this language (the schema of their access point) to describe their resources and to pose queries. The client-peers that use their own defined schema (or simply a standard schema but different than the one that their access point supports) use this global (or middleware) language in order to be able to communicate with the rest of the client-peers by "translating" their metadata items and queries into the schema of their access point as shown in Figure 4.1.

An important characteristic is that there is no global schema predefined both for client-peers and server-peers. The schema of a server-peer is defined by the administrator of the network, when the server-peer bootstraps and remains the same, while the server-peer is connected to the network. Only the administrator has the authority to update the schema of a server-peer, for example, in order to support the schema of new client-peers. Thus, we can create *clusters* where client-peers connect to a server-peer that its schema supports a specific *topic*. The schema of a server-peer is stored locally in the *serverSchema* data structure as described in Section 3.2.1.

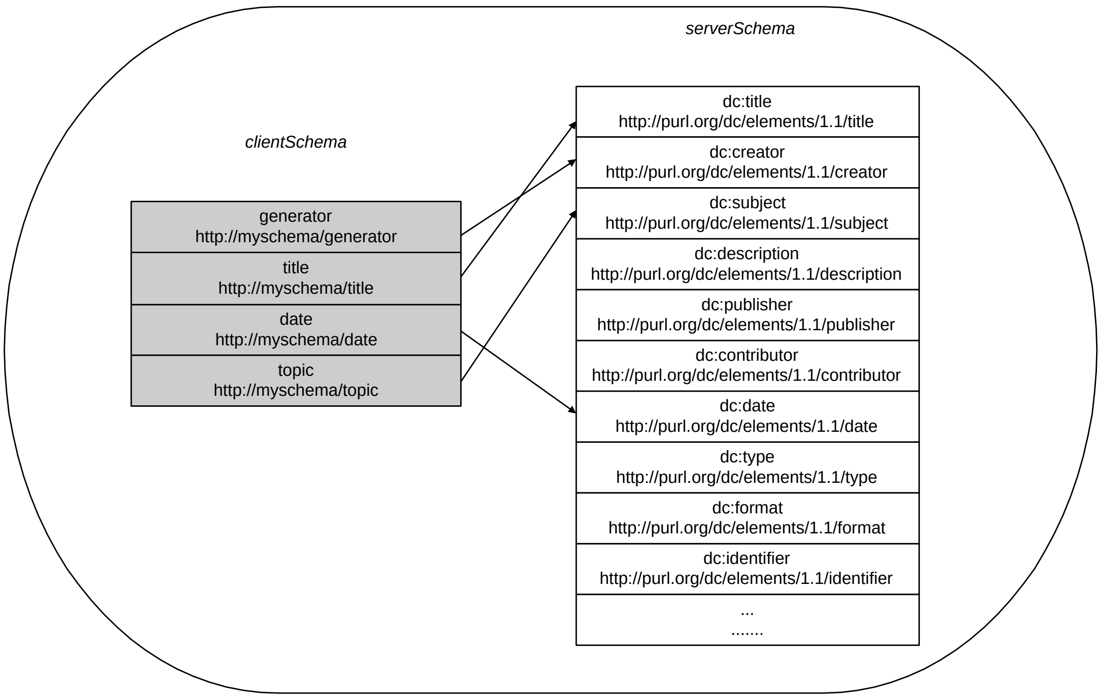

serverSchema
clientSchema
generator
http://myschema/generator
title
http://myschema/title
date
http://myschema/date
topic
http://myschema/topic
dc:title
http://purl.org/dc/elements/1.1/title
dc:creator
http://purl.org/dc/elements/1.1/creator
dc:subject
http://purl.org/dc/elements/1.1/subject
dc:description
http://purl.org/dc/elements/1.1/description
dc:publisher
http://purl.org/dc/elements/1.1/publisher
dc:contributor
http://purl.org/dc/elements/1.1/contributor
dc:date
http://purl.org/dc/elements/1.1/date
dc:type
http://purl.org/dc/elements/1.1/type
dc:format
http://purl.org/dc/elements/1.1/format
dc:identifier
http://purl.org/dc/elements/1.1/identifier
...
....

Figure 4.2: An example of a possible mapping between the schema of a client-peer and the one of a server-peer

4.1.2 The schema supported by client-peers

Each client-peer in the network supports a schema. The schema that a client-peer c supports is the return value of the function $clientSchema(c)$. The schema of a client-peer is a set of attributes and its usage is that the client-peer uses these attributes in order to describe its resources (create metadata items) and to pose queries. A client-peer c that is connected to a server-peer x, adopts by default the schema that x supports. This means, that the schema $clientSchema(c)$ becomes equal to the schema $serverSchema(x)$. For example, in the case that the schema $serverSchema(x)$ is the whole Dublin Core element set, then c is going to support the DC schema too. This happens when a client-peer connects to a server-peer. The server-peer sends its serverSchema data structure and the client-peer clones it and stores it as the local $accesspointSchema$ data structure, which is identical to the serverSchema data structure as described in Section 3.2.1.

Until now we have described how a client-peer adopts the schema of its access point. Our system goes a step further and provides to client-peers a further opportunity in terms of the selection of the schema that they are going to support. Client-peer is not obligated to accept and support the schema that is supported by its access point. On the contrary, a client-peer can use any schema different than the one its access point supports. This is a useful extension. Consider, for example, the case that a library connects in order to make available its books and journals. If the library has already metadata for its resources, described in a schema different than the schema of the server-peer, then it can use this feature of the system and make its resources available to other client-peers in a transparent way without having to create new metadata items. A client-peer c attached to an access point x that uses a schema different that the schema $serverSchema(x)$:

1. designates the component attributes that represent the schema $clientSchema(c)$, namely it denotes a *name* and a *URI expression* for each attribute.
2. Maps the schema $clientSchema(c)$ to the schema $serverSchema(x)$.

Note, that the mapping between these two schemas, actually refers to the semantic connection of each attribute, which belongs to $clientSchema(c)$, to one attribute of the schema $serverSchema(x)$.

The schema of a client-peer (that uses a schema different than the one of its access point) is stored locally in the *clientSchema* data structure. This data structure is a java hash table and uses the name of the attribute as input to the hash function. Each record contains the name and the URI expression of the attribute along with a pointer. The pointed record belongs to the $serverSchema$ data structure.

The basic rule that a client-peer must follow, so as the process of the mapping to be efficient, is that it must map all attributes of its schema to one attribute of the schema that is supported by its access point. If it does not map all the attributes of its schema, then the research for matching resources will be thorniness.

A possible example of mapping between the schema of a client-peer and the one of a server-peer is shown in Figure 4.2. Consider a client-peer c and its access point x. Assume that x supports the whole Dublin Core metadata element set, and that c configures its schema using the following four attributes:

name	URL
generator	http://myschema/generator
title	http://myschema/title
date	http://myschema/date
topic	http://myschema/topic

As it is shown in Figure 4.2, the mapping seem to be successful, namely the meaning of the elements that are mapped is similar. In particular, the meaning of the word “generator” matches better with the attribute “dc:creator” than any other attribute of the $serverSchema(x)$, seeing that the words “generator” and “creator” imply the entity that is responsible for the making of a resource (for our application, this entity could be a person, a services or an organization). In addition, the meaning of the word “topic” matches to the attribute “dc:subject”, because these two attributes refer to the centric idea that describes a resource. Also, the attribute “date” and “title” match to the attributes “dc:date” and “dc:title” respectively.

Whether the mapping, among the attribute of the schemata, responds to the objective reality, namely to the common way of perception or not, it depends on the discernment of the user, who uses the client-peer application program. We do not use an automatic mechanism for checking the mapping; instead we assume that each client-peer understands the lexical meaning of the elements in the same or a similar way that the other client-peers of the system do. This means, that the client-peer, of the above example, can map the attribute “generator” to any attribute of the $serverSchema(x)$, as for example to the attribute “dc:subject”. However, this mapping is not compatible with the global knowledge, namely the “generator” is someone that is responsible for a creation, while the word “subject” denotes the topic of the content of something (in our system, the topic of a resource), so these two attribute do not match semantically. In the case that the client-peers create unseasonable mapping, the resources that are going to retrieved by searching, will be irrelevant to the requests.

At this point let us give the definition of the *current schema* of a client-peer. The current schema of a client-peer c is the return value of the function $currentSchema(c)$. The usage of the current schema comes out from the fact that a client-peer may support a schema but only use a subset of it to describe its resources. The system will work more efficiently if the server-peer knows at all times the schema that the client-peers actually use and not the schema that they support. Each attribute of the current schema of a client-peer is associated with an integer. This integer denotes the number of resources for which the attribute has been used in order to create metadata items. For an attribute a we call this number the frequency of a, denoted by $frequency(a)$. For a client-peer c, stands $currentSchema(c) \subseteq clientSchema(c)$.

4.2 Network Functionalities

In the previous sections we presented a general discussion about the schema that different types of peers understand and support. In this section we analyze the functionalities of the system and how the schema of client-peers is modulated during the execution of these basic operations. We will describe in detail what happens when a client-peer:

- publishes a resource.
- updates the metadata of a resource.

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 - <rdf:Description rdf:about="The Lake.wav">
     <dc:creator>Philip Glass</dc:creator>
     <dc:title>The Lake</dc:title>
     <dc:date>1993</dc:date>
   </rdf:Description>
 </rdf:RDF>
```

Table 4.1: XML representation of the $metadata(r)$

- removes a resource.

We describe how these functionalities are executed using a schema-based routing way and how impress the schema that is supported by other peers of the system.

4.2.1 Publishing resources

As we have already mentioned, a client-peer is able to share resources. In order to share a resource r a client-peer has to *publish* r. In our current implementation (file sharing application) a resource can be a file of any type located locally in the client-peer. A client-peer and its access point, follow a schema-based routing way, in order to publish a resource.

In order for the user of the client-peer c application to publish a resource, the above procedure is followed:

1. The user of client-peer c application chooses to publish a resource r.

2. The user fills in the metadata that describe the resource r. Thus, a metadata item, denoted by $metadata(r)$, is created. This metadata item (attribute value set) consists of attributes contained in the schema $clientSchema(c)$.

3. The metadata item of r is stored as triples (one for each attribute), in the local MySQL database of c.

4. If client-peer c is a normal one, it forwards the metadata item to its access point and the procedure stops.

5. If client-peer c is a volunteer, it checks which attributes of $metadata(r)$, are included in the schema $currentSchema(c)$.

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 - <rdf:Description   rdf:about="E:/IntelliJ-IDEA-3.0.1/src/client/2000/Schema.xml">
     <dc:creator>1</dc:creator>
     <dc:title>1</dc:title>
     <dc:date>1</dc:date>
   </rdf:Description>
 </rdf:RDF>
```

Table 4.2: The modulation of $currentSchema(c)$ after the publish of r

6. For each attribute a such as that $a \in metadata(r)$ and $a \in currentSchema(c)$, c updates $currentSchema(c)$, by increasing by one the number that denotes the frequency of the attribute a.

7. For each attribute a such as that $a \in metadata(r)$ and $a \notin currentSchema(c)$, c updates $currentSchema(c)$, by adding attribute a in $currentSchema(c)$ and setting, the frequency of the attribute a to one.

In the case, that c adds new attributes to the schema $currentSchema(c)$, informs its access point x on this change, by forwarding the updated $currentSchema(c)$ to x. This happens because the access point x is interested in which attributes are used by c and not in how many times c has used each attribute. This information is important for the process of answering queries[1]. The algorithm for the procedure of publishing a resource is shown below.

[1]This process is described in Section 4.3.

Algorithm. Publishing a resource

Suppose that c is the client-peer, x is the server-peer and r is the resource for publish.

```
1: if $normal == True$ then
2:     $send(\text{“publish”}, x, metadata(r))$;
3:     return;
4: $updated = False$;
5: for all $a$ in $metadata(r)$ do
6:     if $a$ in $currentSchema(c)$ then
7:         $frequency(a) = frequency(a) + 1$;
8:     else if $a$ not in $currentSchema(c)$ then
9:         $currentSchema(c).add(a)$;
10:        $frequency(a) = 1$;
11:        $updated = True$;
12: if $updated == True$ then
13:    $send(\text{“update schema”}, x, currentSchema(c))$;
```

Let us make clear the process of publishing a resource, by describing an example. Assume that a server-peer x supports the Dublin Core element set (DC), namely $serverSchema(x) = DC$. Then, a volunteer client-peer c, connects to x. The client-peer c requests from x the schema $serverSchema(x)$ and sets $clientSchema(c) = serverSchema(x)$. This means that c becomes able to understand and communicate with other peers using the DC schema. In addition, assume that the user of the client-peer c application uses the application for first time, thus he has not published any resources yet. When the user chooses to publish a resource r, he describes it by setting values to the attributes of the schema $clientSchema(c)$. He is not fain to use all the attributes that constitutes the schema $clientSchema(c)$. Actually, he can use whichever subset of it. Assume that the user of our example selects to publish a resource r, named “The Lake.wav” and describes it in the following way:

attribute	value
dc:creator	Philip Glass
dc:title	The Lake
dc:date	1993

Then, the metadata item of r is created and contains the attribute value pairs

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 - <rdf:Description rdf:about="Symphony No.1">
    <dc:creator>Gustav Mahler</dc:creator>
    <dc:title>Symphony No.1 - Songs of a wayfarer</dc:title>
    <dc:subject>classic music</dc:subject>
  </rdf:Description>
 </rdf:RDF>
```

Table 4.3: XML representation of the *metadata*(r')

that were used for the description of r. An XML representation of the metadata item of r is shown in Table 4.1.

Each attribute value pair in the metadata item of r is stored as a triple, in the local MySQL database of c, in the form of $subject[predicate \rightarrow object]$. In our example the following three triples are inserted in the MySQL database.

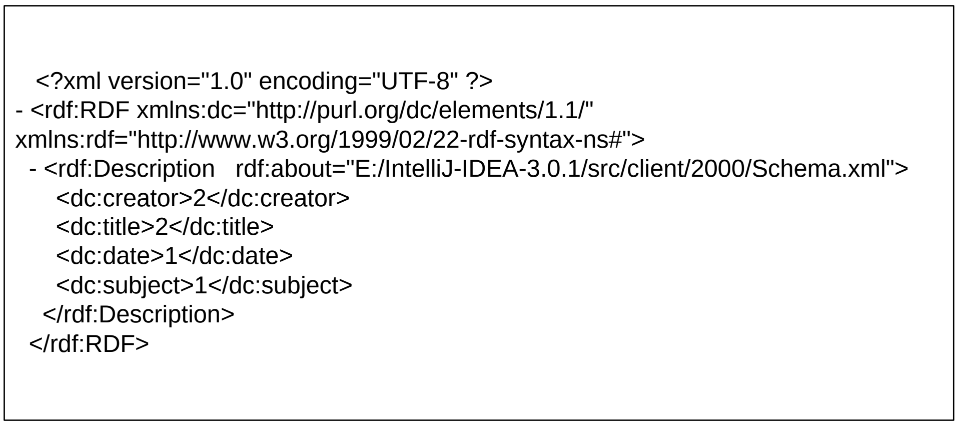
<"http://The Lake.wav", "http://purl/org/dc/elements/1.1/creator", "Philip Glass">
<"http://The Lake.wav", "http://purl/org/dc/elements/1.1/title", "The Lake">
<"http://The Lake.wav", "http://purl/org/dc/elements/1.1/date", "1993">

In our example, the schema $currentSchema(c)$ is initially empty, because the user of the client-peer c application participates in the system for the first time, so he has not published any resources yet. After the resource r has been published, the schema $currentSchema(c)$ is going to become as it is shown in Table 4.2. Since the current schema has been updated, c forwards to x the schema $currentSchema(c)$.

Let us now consider another example, so as to make more clear the process that a client-peer follows, when publishing a resource. In this example we assume that the same client-peer c publishes a resource r' that is named "Symphony No.1" and describes it, by setting the following values to the attributes:

attribute	value
dc:creator	Gustav Mahler
dc:title	Songs of a wayfarer
dc:subject	classic music

Then, the metadata item of r is created by c. An XML representation of this



```
  <?xml version="1.0" encoding="UTF-8" ?>
- <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  - <rdf:Description   rdf:about="E:/IntelliJ-IDEA-3.0.1/src/client/2000/Schema.xml">
      <dc:creator>2</dc:creator>
      <dc:title>2</dc:title>
      <dc:date>1</dc:date>
      <dc:subject>1</dc:subject>
    </rdf:Description>
  </rdf:RDF>
```

Table 4.4: The modulation of *currentSchema*(*c*) after the publish of *r'*

metadata item is shown in Table 4.3. Then, c inserts the metadata item of r' in the local MySQL database, which then is going to include the total information:

<"http://The Lake.wav", "http://purl/org/dc/elements/1.1/creator", "Philip Glass">
<"http://The Lake.wav", "http://purl/org/dc/elements/1.1/title", "The Lake">
<"http://The Lake.wav", "http://purl/org/dc/elements/1.1/date", "1993">
<"http://Symphony No.1", "http://purl/org/dc/elements/1.1/creator", "Gustav Mahler">
<"http://Symphony No.1", "http://purl/org/dc/elements/1.1/title", "Symphony No.1 - Songs of a wayfarer">
<"http://Symphony No.1", "http://purl/org/dc/elements/1.1/subject", "classic music">

The next step that the client-peer c must take, is to modulate the schema $currentSchema(c)$. Client-peer c checks which attributes of $metadata(r')$ are included in the $currentSchema(c)$ and which do not. The attributes "dc:creator" and "dc:title" are already in the $currentSchema(c)$ as we can see in the Table 4.2, so c increases the frequency of these two attributes, by one. The attribute "dc:subject" is used, for the first time for the description of a resource, so c adds it to the $currentSchema(c)$ and sets its frequency to one. Therefore, the schema $currentSchema(c)$ is updated and becomes as it is shown in Table 4.4. Because of the addition of one new attribute (of the "dc:subject") to the $currentSchema(c)$, c informs the server-peer x on this change, namely c forwards to x its updated schema $currentSchema(c)$.

4.2.2 Removing resources

A client-peer can *remove* a resource that has already published. This is useful in case that a resource is no longer valid. As we will see this procedure affects the schema of the client-peer.

When a client-peer c that is connected to a server-peer x, selects to remove a resource r, the following process takes place:

1. The GUI of the client-peer c application presents to the user, all the resources that he has published.

2. The user selects a resource r to remove.

3. Then, client-peer c connects to the local MySQL database and:

   (a) retrieves the attributes that were used for the description of r. The returned attributes are stored on a temporary item, denoted by *attributes(r)*. The RDQL query that c queries its local MySQL database is:

   ```
   SELECT ?attribute
   WHERE (r, ?attribute, ?value)
   ```

   (b) deletes the metadata item of r from the database.

4. If client-peer c is normal, it informs its access point that it removed r and the function returns.

5. If client-peer c is a volunteer one, it modulates the schema $currentSchema(c)$, according to the attributes that the item *attributes(r)* is consisted of. In particular, for each attribute a, such as $a \in attributes(r)$, the frequency of a is decreased by one, namely $frequency(a) = frequency(a) - 1$. In the case that the frequency of an attribute becomes equal to zero, then this attribute is removed from the schema $currentSchema(c)$.

6. In the case that n attributes are removed from the schema $currentSchema(c)$, where $n \geq 1$, then the client-peer c sends to its access point x the updated $currentSchema(c)$. On the contrary, if $n = 0$ (no attribute is removed from the schema $currentSchema(c)$), then c does not inform x on the fact that the schema $currentSchema(c)$ is updated.

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 - <rdf:Description   rdf:about="E:/IntelliJ-IDEA-3.0.1/src/client/2000/Schema.xml">
     <dc:creator>2</dc:creator>
     <dc:title>2</dc:title>
     <dc:subject>3</dc:subject>
     <dc:format>1</dc:format>
     <dc:date>1</dc:date>
     <dc:language>2</dc:language>
   </rdf:Description>
 </rdf:RDF>
```

Table 4.5: The $curentSchema(c)$ before the resource r is removed

Algorithm. Removing a resource

Suppose that c is the client-peer, x is the server-peer, r is the resource for remove.

```
1: if $c == normal$ then
2:     $send(``remove", x, r)$;
3:     return;
4: $updated = False$;
5: for all $a$ in $attribute(r)$ do
6:     if $a$ in $currentSchema(c)$ then
7:         $frequency(a) = frequency(a) - 1$;
8:         if $frequency(a) == 0$ then
9:           $currentSchema(c).remove(a)$;
10:          $updated = True$;
11: if $updated == True$ then
12:    $send(``update\ schema", x, currentSchema(c))$;
```

Let us now give an example to make the process of removing a resource more clear. Assume that a client-peer c is connected to a server-peer x and that both of them support the Dublin Core element set schema, namely $serverSchema(x) = DC$ and $clientSchema(c) = DC$. Assume that c has already published resources and that its schema $currentSchema(c)$ is as it is shown in Table 4.5.

Assume, that the client-peer c has published a resource r using the metadata item of Table 4.6. Now, client-peer c wants to remove r. Client-peer c connects

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 - <rdf:Description rdf:about="The Chatty Web">
     <dc:creator>Karl Aberer</dc:creator>
     <dc:title>The Chatty Web: Emergent Semantics Through Gossiping</dc:title>
     <dc:subject>Semantic Web</dc:subject>
     <dc:date>2003</dc:date>
     <dc:language>En</dc:language>
   </rdf:Description>
 </rdf:RDF>
```

Table 4.6: The metadata file of the resource r

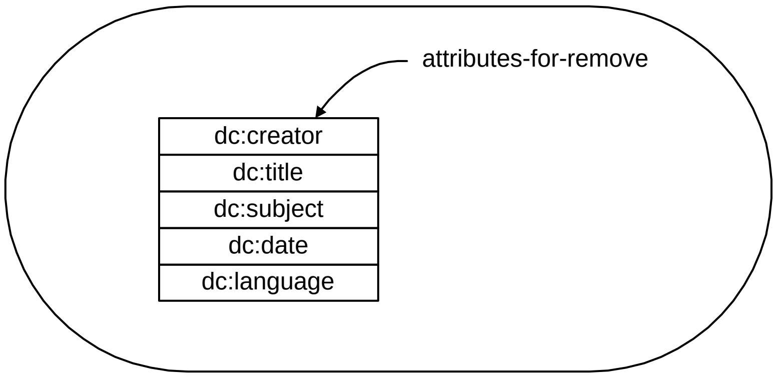


Figure 4.3: The temporary item attributes

to the local MySQL database and queries, with an RDQL query, to find out the attributes of the schema $clientSchema(c)$ that were used for the description of r when publishing it. The returned attributes are stored on the *attributes temporary* item, denoted by *attributes(r)*. A view of $attribute(r)$ is shown in Figure 4.3. Then, c deletes the metadata item of r, from the local database.

The next step that the client-peer c takes, is to modulate the schema $currentSchema(c)$. For this purpose, c uses the item $attributes(r)$ and the schema $currentSchema(c)$. For each attribute a such as $a \in attribute(r)$, c decreases per one the frequency of a, $frequency(a) = frequency(a) - 1$. For example, because the attribute "dc:creator" exists in $attributes(r)$, c sets its frequency to "1". For the attributes "dc: title", "dc:subject" and "dc:language", also included in $attribute(r)$, their frequency is set to "1", "2" and "1", respectively. For the attribute "dc:date", included in $attributes(r)$, its frequency is reduced to zero. Thus, c removes the attribute "dc:date" from the schema $currentSchema(c)$, which becomes as it is shown in Table 4.7. Because of the deletion of the attribute "dc:date", c sends to

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 - <rdf:Description   rdf:about="E:/IntelliJ-IDEA-3.0.1/src/client/2000/Schema.xml">
     <dc:creator>1</dc:creator>
     <dc:title>1</dc:title>
     <dc:subject>2</dc:subject>
     <dc:format>1</dc:format>
     <dc:language>1</dc:language>
   </rdf:Description>
 </rdf:RDF>
```

Table 4.7: The $curentSchema(c)$ after the remove of the resource r

its access point x, the updated schema $currentSchema(c)$. This is necessary to happen because the server-peer x should know momentarily, which schema is used by its client-peers, so as to forward queries to the appropriate client-peers. In the case that c deletes the attribute "dc:date" from the schema $currentSchema(c)$, but does not send the updated schema to its access point x, then x will continue to send queries to c, which demand the attribute"dc:date", but c will not have related resources.

4.2.3 Updating the metadata of resources

A client-peer can update the metadata item of a resource that it has previously published. This scenario has been designed in order for the search and the match of resources, to become more efficient. More precisely, for a resource r that a client-peer has published, the client-peer can:

- *change* the value of one or more of the attribute value pairs that have been used for the description of r.
- *add* one or more new attribute value pairs in $metadata(r)$, in order to describe the resource r more analytically.
- *delete* one or more attribute value pairs that are not valid for r any more.

Similarly to what happens when publishing or removing a resource, the process of updating a resource is executed in a schema based way too. This means, that the process of updating a resource affects the schema information of the owner client-peer and its access point.

Actually, the functionality of updating a resource, can be seen as the merge of the two basic functionalities of publishing and removing resources. When, a client-peer c, attached to an access point x, selects to update the metadata item of a resource r, the following process takes place:

1. The client-peer c connects to the local MySQL database and requests the metadata of resource r.
2. Client-peer c stores all the attributes that were used for the description of r in a temporary item, denoted by $previous(r)$.
3. Then, the metadata item of r is deleted from the database.
4. The client-peer c application, through the GUI, presents to the user the metadata item of r.
5. The user fill in the form, by adding, removing or updating one or more attribute value pairs.
6. A new metadata item is created for r and stored as triples (one for each attribute) in the database.
7. Then, c modulates the schema $currentSchema(c)$ in the following way:
    (a) For each attribute a such as $a \in previous(r)$ and $a \notin metadata(r)$, the frequency of attribute a is decreased by one, namely $frequency(a) = frequency(a) - 1$. If the frequency of an attribute becomes zero, then this attribute is removed from the current schema of client-peer c.
    (b) For each attribute a such as $a \in metadata(r)$ and $a \notin previous(r)$, if $a \in currentSchema(c)$, then the frequency of a is increased by one or else attribute a is added to the current schema and its frequency is initialized to one. The first case is when an attribute has already been used for another resource so we just increase its frequency, while the second one is when an attribute is used for the first time.
8. In the case that one or more attributes are added or removed from the current schema of client-peer c, c forwards the updated schema $currentSchema(c)$ to its access point.

The algorithm is shown as pseudocode below:

Algorithm. Updating a resource

Suppose that c is the client-peer, x is the server-peer, r is the resource for update.

```
1: if $c == normal$ then
2:     $send(\text{“}update\text{”}, x, r, metadata(r))$;
3:     return;
4: $updated = False$;
5: for all $a$ in $previous(r)$ do
6:     if $a$ not in $metadata(r)$ then
7:         $frequency(a) = frequency(a) - 1$;
8:         if $frequency(a) == 0$ then
9:             $currentSchema(c).remove(a)$;
10:            $updated = True$;
11: for all $a$ in $metadata(r)$ do
12:    if $a$ not in $previous(r)$ then
13:        if $a$ in $currentSchema(c)$ then
14:            $frequency(a) = frequency(a) + 1$;
15:        else if $a$ not in $currentSchema(c)$ then
16:            $currentSchema(c).add(a)$;
17:            $frequency(a) = 1$;
18:            $updated = True$;
19: if $updated == True$ then
20:    $send(\text{“}update\ schema\text{”}, x, currentSchema(c))$;
```

Let us now give an example to make the process of updating a resource more clear. Assume that a client-peer c is connected to a server-peer x. The current schema of client-peer c is shown in Table 4.5. We assume that the user of client-peer c application, selects to update a resource r that is named “The Chatty Web”. The client-peer c connects to the local database and selects from the mass of information all the triples that refer to r, namely:

<“http://The Chatty Web”, “http://purl/org/dc/elements/1.1/creator”, “Karl Aberer”>
<“http://The Chatty Web”, “http://purl/org/dc/elements/1.1/title”, “The Chatty Web: Emergent Semantics Through Gossiping”>
<“http://The Chatty Web”, “http://purl/org/dc/elements/1.1/subject”, “Semantic Web”>
<“http://The Chatty Web”, “http://purl/org/dc/elements/1.1/date”, “2003”>
<“http://The Chatty Web”, “http://purl/org/dc/elements/1.1/language”, “En”>

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  - <rdf:Description rdf:about="The Chatty Web">
      <dc:creator>Karl Aberer, Philippe Cudre-Mauroux</dc:creator>
      <dc:title>The Chatty Web: Emergent Semantics Through Gossiping</dc:title>
      <dc:subject>Semantic Web</dc:subject>
      <dc:publisher>WWW2003</dc:publisher>
      <dc:language>En</dc:language>
    </rdf:Description>
  </rdf:RDF>
```

Table 4.8: The new metadata that c sets while updating r

Client-peer c stores in the temporary item *previous*(r), all the attributes that were used for the description of r, namely *previous*(r) contains the attributes: “dc:creator”, “dc:title”, “dc:subject”, “dc:date” and “dc:language”. Client-peer c deletes the above triples from the local database. The user of client-peer c through the GUI defines the new metadata item of r. Assume that the new metadata item of r is as is shown in Table 4.8.

The next step that c takes, is to modulate the schema *currentSchema*(c). For this purpose, c access the schema *currentschema*(c) and the items *metadata*(r) and *previous*(r). For each attribute $a \in$ *previous*(r) and $a \notin$ *metadata*(r), c decreases per one the frequency of a. In this case goes the attribute “dc:date” that was used for the description r but while updating the user did not set a value to it. The frequency of the attribute “dc:date” was “1”, so the client-peer c deletes it from the schema *currentSchema*(c). Client-peer c checks another case, if there are attributes that are included in the item *metadata*(r) but are not included in the item *previous*(r), namely if, while updating of r, the user uses attributes that had not used before. The attribute “dc:publisher” belongs to this case. Client-peer c adds this attribute in the schema *currentSchema*(c) and set its frequency to one. The rest of the attributes of item *metadata*(r), are included in the term *previous*(r), so do not affect the current schema of c. In this example, the update of the resource r, affects the schema *currentSchema*(c), because of the deletion of an attribute (“dc:date”) and the addition of another (“dc:publisher”), which is shown in Table 4.9. Because of that, the client-peer c forwards its updated current schema to x.

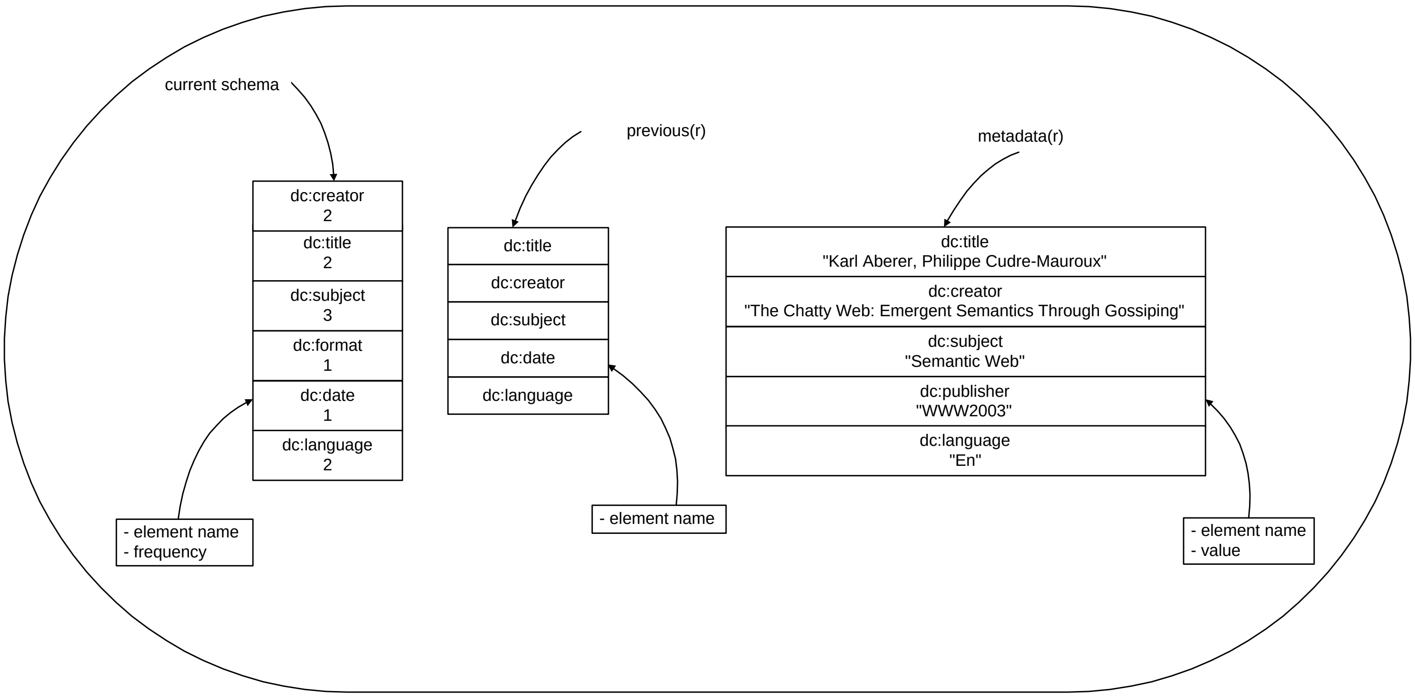


Figure 4.4: The current schema and the temporary items, *previous(r)* and *metadata(r)*

4.3 The process of answering queries

In the previous sections we analyze the way that a client-peer publishes, updates and removes resources. The next logical step is to define the way a query is posed and answered, which is what we study in this section.

A client-peer searches for resources with specific features. A client-peer describes the resources to be retrieved with an RDQL query. In the query, the client-peer determines which attribute value pairs or combination of attribute value pairs should outline the requested resources. Note, that the expressivity of the query is limited only by the RDQL query language. We assume that the user of the application are expert users that are able to build RDQL queries.

A client-peer poses a query to the system by forwarding it to its access point. The latter, using the schema-information that has stored and metadata items, takes the appropriate actions in order to produce answers. At this point, recall that a server-peer manages metadata items of normal client-peers and schema information (current schema) of volunteer client-peers. Thus, for a *complete* set of answers both metadata items in the server-peer and metadata items of volunteer client-peers must be accessed. The former is satisfied by posing the query to the local databases of the server-peer, while the latter is satisfied by forwarding the query to those client-peers that *may* have matching resources, namely to those

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  - <rdf:Description   rdf:about="E:/IntelliJ-IDEA-3.0.1/src/client/2000/Schema.xml">
      <dc:creator>2</dc:creator>
      <dc:title>2</dc:title>
      <dc:subject>3</dc:subject>
      <dc:format>1</dc:format>
      <dc:publisher>1</dc:publisher>
      <dc:language>2</dc:language>
    </rdf:Description>
  </rdf:RDF>
```

Table 4.9: The modulation of the current schema of a client-peer after the updating of a resource

client-peers that the attributes in their current schema match the attributes that were used for building the query. The steps that are followed when a client-peer c, attached to an access point x, poses a query q are the following:

1. The user of the client-peer c application writes an RDQL query.
2. The client-peer c forwards the query q to x. A query message contains the actual query and the IP address and the port of the owner of the query.
3. The server-peer x parses the query q, in order to finds which attributes are used. It does not care about the values that the client-peer has set to the attributes, but just for the actual attributes. A temporary item is created, denoted by $attributes(r)$.
4. For each record k in the local active-clients data structure, the current schema of the client-peer $k.client$ that record k represents is checked against $attributes(r)$ and if there is a match (all attributes of $attributes(r)$ are included in the current schema of k.client) then the query is forwarded to k.client.
5. The server-peer poses the query to its local database and generates zero or more answers. The answers are forwarded directly to client-peer c. An answer contains the metadata item of the matching resource and the IP address and the port of the resource owner client-peer.
6. Each client-peer that receives a query from its access point poses the query to its local database. If it finds one or more matching resources, it generates

the appropriate answers (using the same format as the server-peer) and sends them directly to the client-peer *c* (the IP address and the port of *c* were included in the query).

7. Then, the client-peer *c* can go ahead and request a matching resource.

The example that it is described below, illuminates the functionality of answering query. Assume that a client-peer *c*, which has IP address: 220.17.173.258 and Port : 2000, is connected to a server-peer *x*. The schema that it is supported by *c* is the same schema that is supported by *x*, and it is the Dublin Core schema. The client-peer *c* poses the following RDQL query to *x*:

```
SELECT ?x
WHERE (?x, <http://purl.org/dc/elements/1.1/title>, "RDF"),
      (?x, <http://purl.org/dc/elements/1.1/creator>, "Bill")
```

In this query, *c* searches for a resource that in its title includes the term “RDF” and has as creator someone who is named “Bill”.

The server-peer *x* that receives this query, parses it so as to find which schema-attributes are used. The attributes are the “dc:title” and the “dc:creator”. Then, the server-peer access the data structures *active-clients* and *serverSchema* so as to designate which client-peers include these attributes to their current schema. The two data structures are shown in Figure 4.5. The first client-peer that there is in the data structure *active-clients* (with IP address: 213.16.184.224 and Port: 4838) uses for the description of its published resources, the attributes “dc:title” and “dc:creator”. Thus, the server-peer forwards to this client the above RDQL query. The second client-peer (IP address: 215.17.179.521, Port : 4256), includes in its current schema the attributes “dc:title” and “dc:description”, but it does not include the “dc:creator”, so this client-peer does not have resources that match the query, therefore the server-peer does not forwards the query to this one. The third client-peer (IP address: 220.17.173.258, Port : 2000) is the one that poses the query, so the server-peer does not check if it satisfies the conditions so as to answer the query. The next client-peer (IP address: 222.18.183.253, Port : 4832) uses for the description of its published resources the attributes “dc:creator” and “dc:description”, but does not use the “dc:title” so the server-peer does not forward the query to it.

After the analytical check, the server-peer *x* forwards the query to the client-peers *c1* (IP address: 213.16.184.224, Port: 4838) and *c2* (IP address: 223.20.203.538, Port : 4428). Then, each client-peer queries its local MySQL database. In the case that the client-peer *c1* has published a resource that satisfies the query, *c1* connects to the requester client-peer *c* as to inform it for the answer. It sends to *c* the metadata item of the resource that matches the query. Then, the client-peer *c* decides if it wants to request it or not. In the case that *c* wants the resource,

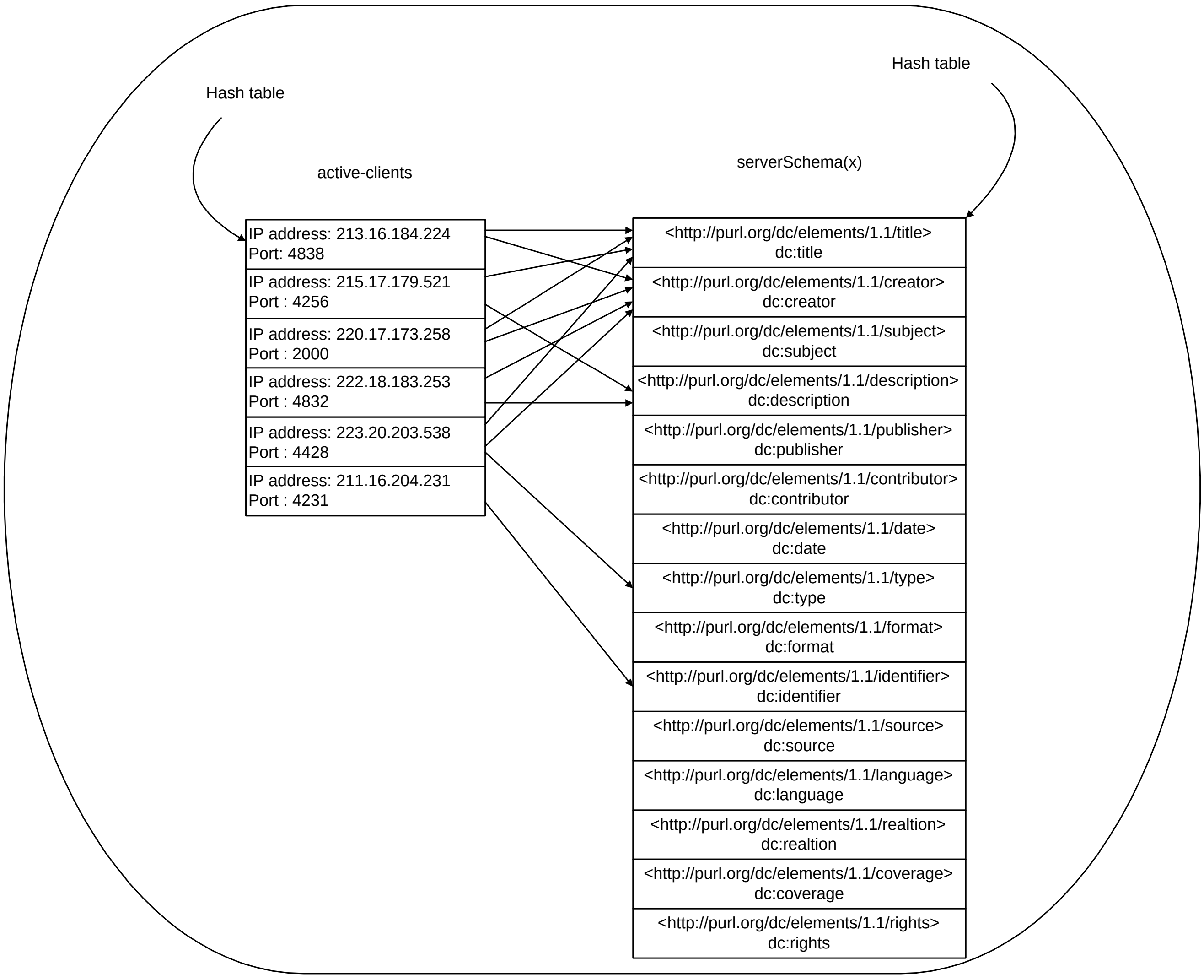

Hash table
Hash table
active-clients
serverSchema(x)
IP address: 213.16.184.224
Port: 4838
IP address: 215.17.179.521
Port : 4256
IP address: 220.17.173.258
Port : 2000
IP address: 222.18.183.253
Port : 4832
IP address: 223.20.203.538
Port : 4428
IP address: 211.16.204.231
Port : 4231
<http://purl.org/dc/elements/1.1/title>
dc:title
<http://purl.org/dc/elements/1.1/creator>
dc:creator
<http://purl.org/dc/elements/1.1/subject>
dc:subject
<http://purl.org/dc/elements/1.1/description>
dc:description
<http://purl.org/dc/elements/1.1/publisher>
dc:publisher
<http://purl.org/dc/elements/1.1/contributor>
dc:contributor
<http://purl.org/dc/elements/1.1/date>
dc:date
<http://purl.org/dc/elements/1.1/type>
dc:type
<http://purl.org/dc/elements/1.1/format>
dc:format
<http://purl.org/dc/elements/1.1/identifier>
dc:identifier
<http://purl.org/dc/elements/1.1/source>
dc:source
<http://purl.org/dc/elements/1.1/language>
dc:language
<http://purl.org/dc/elements/1.1/realtion>
dc:realtion
<http://purl.org/dc/elements/1.1/coverage>
dc:coverage
<http://purl.org/dc/elements/1.1/rights>
dc:rights

Figure 4.5: The data structures *active-clients* and *serverSchema(x)*

it sends a request message to the client-peer *c1* and client-peer *c1* sends the resource.

4.4 When client-peers define their own schema

In the previous sections we studied the processes of publishing, removing and updating resources. We also discussed about the way queries are handled. We analyzed the distinct steps of these processes and the way they modulate the schema information of client-peers and server-peers. In the previous section we focused on the scenario where the client-peers support the schema of their access point. In this section we analyze how the functionalities of our system are executed, if a client-peer supports a schema different than the one of its access point.

In the case that a client-peer c is connected to a server-peer x and selects to define its own schema, it follows the process that we describe in Section 4.1.2, namely it defines the component attributes of its schema $clientSchema(c)$ and maps each attribute to one of the attributes of the schema $serverSchema(x)$. When the client-peer selects to use one of the functionalities of the system (publish, remove, update, query), it follows the procedures that we describe in Sections 4.2.1, 4.2.2, 4.2.3 and 4.3. The additional step is the execution of the extensional process of *translation*. By the term translation we mean the mapping between the schema of a client-peer and the schema that is supported by its access point, so as to change each attribute that belongs to one schema by its relevant attribute of the other schema. The process of translation has to be executed each time a client-peer:

- sends to access point an updated version of its current schema (stands for volunteer client-peers only).
- publishes or updates a resource and sends to its access point the metadata item of the resource (stands for normal client-peers only).
- poses an RDQL query.
- receives from its access point an RDQL query, so as to answer it.
- receives (either from its access point or from another client-peer) a metadata item as an answer to a query that it posed.

The process of translation is critical and necessary, otherwise the communication between the peers of the system will be inefficient, because each peer understands just the schema that it supports and does not understand the schema of other peers. At this point, recall that a volunteer client-peer does all the translations on its own while a normal client-peer does not do any translation at all.

The access point of a normal client-peer does this job. This means, that a volunteer client-peer translates all messages before sending them to its access point (for example, a query message or an update schema message) or to other client-peers (for example, an answer message). A normal client-peer sends its metadata items (publish, remove or update resource messages) and queries untranslated. The access point translates those messages as long as it receives them. In addition, the access point translate an answer before sending it to a normal client-peer. Note, that the only case that a normal client-peer does a translation on its own, is when receiving an answer from another client-peer, namely a volunteer one.

Let us now give an example in order to explain how the process of publishing a resource is executed, in the case that a client-peer has defined its own schema. The scenarios where a client-peer removes or updates a resource are of similar logic, and we will omit them. Assume that a client-peer c participates in the system using a different schema than the one that its access point x supports. The server-peer x supports the Dublin Core element set. The mapping between the schema *clientSchemac* and the schema $serverSchema(x)$ is shown in Figure 4.2. The client-peer c participates in the system for the first time, so $currentSchema(c) = \varnothing$. It selects to publish a resource r, named "The Lake.wav" and describes it by setting values to the following attributes:

attribute	value
generator	Philip Glass
title	The Lake
date	1993

Then, the metadata item of r is created, and contains the attribute value pairs that are used for the description of r. An XML representation of the metadata item of r is shown in Table 4.10.

Each attribute value pair in the metadata item of r is stored as a triple, in the local MySQL database of c, in the form of $subject[predicate \rightarrow object]$.

<"http://The Lake.wav", "http://myschema/generator", "Philip Glass">
<"http://The Lake.wav", "http://myschema/title", "The Lake">
<"http://The Lake.wav", "http://myschema/date", "1993">

The publication of the resource r causes the change of the schema $currentSchema(c)$. Before the publish, the schema $currentSchema(c)$ was empty, so now the client-peer adds to it the three attributes (generator, title and date) that are used for the description of the resource and sets the frequency of each attribute's usage to one. The schema $currentSchema(c)$ becomes as is shown in Table 4.11.

Because of the addition of these three attributes to the schema $currentSchema(s)$,

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 - <rdf:Description rdf:about="The Lake.wav">
     <generator>Philip Glass</generator>
     <title>The Lake</title>
     <date>1993</date>
   </rdf:Description>
  </rdf:RDF>
```

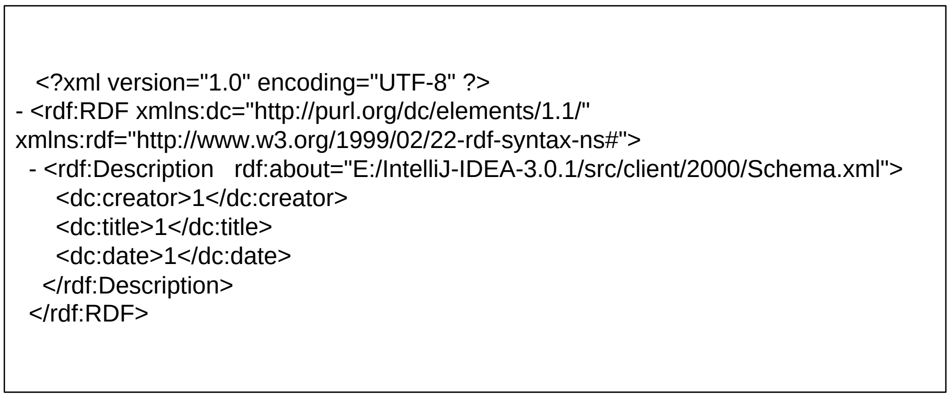
Table 4.10: XML representation of the *metadata*(r), in user-defined schema

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 - <rdf:Description   rdf:about="E:/IntelliJ-IDEA-3.0.1/src/client/2000/Schema.xml">
     <generator>1</generator>
     <title>1</title>
     <date>1</date>
   </rdf:Description>
  </rdf:RDF>
```

Table 4.11: The modification of user-defined current schema after the publish of r

the client-peer should send to its access point the updated schema *currentSchema*(s). The next step that c takes is to translate the *currentSchema*(s). It transforms each attribute of the schema *clientSchema*(c) to the relevant attribute of the schema *serverSchema*(x). In particular, c changes the attribute "generator" to the attribute "dc:creator", the "title" to "dc:title" and the "date" to "dc:date". The XML representation of the changed schema *currentSchema*(c) is shown in Table 4.12. After that, the client-peer sends its schema *currentSchema*(c) to its access point x. The server-peer x, understands and stores to its data structures *active-clients* and *serverSchema*(x) that the specific client-peer c supports the attributes "dc:creator", "dc:title" and "dc:date"; it does not care if c supports the actual attributes or others that it has related to these.

Let us now give an example of what happens when a client-peers that supports a schema different than the one of its access point, poses a query. Consider that the server-peer x supports the Dublin core element set and the mapping between the two schemas is the same as above (shown in Figure 4.2). Client-peer



```
  <?xml version="1.0" encoding="UTF-8" ?>
- <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
 - <rdf:Description   rdf:about="E:/IntelliJ-IDEA-3.0.1/src/client/2000/Schema.xml">
     <dc:creator>1</dc:creator>
     <dc:title>1</dc:title>
     <dc:date>1</dc:date>
   </rdf:Description>
  </rdf:RDF>
```

Table 4.12: The translated current schema

c wants to find resources that have been created by a person named "Mahler", the user of c writes the RDQL query using the schema that he has defined:

```
SELECT ?r
WHERE (r, ?generator, "Mahler")
```

In the case that client-peer c is volunteer, translates the query before it poses it to its access point. Thus it becomes:

```
SELECT ?r
WHERE (?x, <http://purl.org/dc/elements/1.1/creator>, "Mahler" )
```

The server-peer x receives an RDQL query written using its schema, so it follows the process of answering queries as we described in Section 4.3.

In the case that the client-peer is normal, it sends the query untranslated to its access point.

The server-peer x stores the mapping between its schema and the schemas of all its normal client-peers, so when x receives the untranslated query, it uses the mapping between the schema $serverSchema(x)$ and the schema $clientSchema(c)$ and translates the query. Then, it follows exactly the same process as to forward it to the appropriate client-peers.

4.5 Summary

In this chapter we presented the schemas that are supported by different types of system peers. We analyzed the system functionalities and how these are executed in a schema-based routing way. We presented the four distinct scenarios of the system, which are modulated based on the type of client-peer (normal or volunteer) and the client schema (if it is a user-defined or not). In the following chapter, we present some well-known peer-to-peer systems and their architecture.

Chapter 5

Concluding Remarks

The combination of Semantic Web and peer-to-peer technologies provides fast and reliable data retrieval and efficient search. For peer-to-peer environments metadata are absolutely crucial, in order to describe the resources managed by the peers. The RDF-based peer-to-peer networks have a number of important advantages over previous more simple peer-to-peer networks. In this dissertation we have described the design and implementation of an RDF-based, hybrid peer-to-peer network, in which peers provide and use explicit schema description of their content. We also introduce a new distinction of client peers in traditional hybrid peer-to-peer networks. We distinguished the client peers to more and less powerful and defined the role of each type of client peers. The main goal was to build a system to work in the real word, the Internet. So we implement a file sharing application that allow users to define their own metadata vocabulary and then to publish available resources to the network, or to query the system for desirable resources.

In this dissertation, we initially surveyed the area of distributed systems. We presented a taxonomy of computer systems from the peer-to-peer perspective, and then we classify the peer-to-peer systems in terms of their application domain. We briefly discussed some alternative architectures for peer-to-peer systems and the some well-known, representative networks of these architectures. Then, we presented the necessary knowledgeable background for this thesis. We gave a brief description of the RDF data model, the Dublin Core metadata standard, the Jena API, the MySQL database and the RDQL query language.

After, we presented the architecture of our system. We described the functionalities that efficiently are supported using such an architecture. We analyzed the conditions that specify if a client-peer is *normal* or *volunteer*, and then we defined exactly the role that each type of client-peers, has in our network application. Then, we discussed the role of the server-peer and its reactions to requests from client-peers. We mentioned the information that it stores in case that it handles requests of normal or volunteer client-peers, and how it executes each request. Because of we consider that our application intend to work in the real world, we have focused on a fault-tolerance mechanism. In this way, we avoid a number of problems that are brought to the system in the case that a client-peer is silent disconnected or fails.

Then, we discussed what kind of schema information each type of peer (server, normal, volunteer) stores and how this information can be used for routing in our peer-to-peer network. We analyzed the functionalities of the system from the schema information perspective, namely we described the way that each system functionality is executed and how the schema of client-peers is modulated during the execution. We described in detail the algorithms for the basic procedure of our system:

1. The algorithm of *publishing* a resource, which is used by client-peers even if they are normal or volunteer, in order to publish available resources.

2. The algorithm of *removing* a resource, which is used by client-peers in order to remove resources that they have already published.

3. The algorithm of *updating* a resource, which is used by client-peers in order to update the metadata of already published resources.

Finally we described the process of answering queries. We studied the case that a client-peer, which has defined its own schema, sends queries to the system; we analyzed the process of *translation* that is demanded in order to be forwarded and to be answered such a query.

Bibliography

[1] Dublin core metadata web site. http://purl.org/dc/.

[2] The Gnutella Protocol Specification v0.4. Clip2 report, http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.

[3] Dublin core metadata element set: Reference description. http://dublincore.org/documents/dces/, 2003.

[4] JXTA 2001. The JXTA Home Page www.jxta.org.

[5] Microsoft 2001. Microsoft .NET Passport Technical Overview September 2001. www.sec.informatik.tu-darmstadt.de/de/lehre/SS03/itsec2/uebungen/wp_engl_net_passport.pdf.

[6] ISO 8879. *Information processing — Text and Office Systems — Standard Generalized Markup Language (SGML)*. International Organization for Standardization, Geneva, Switzerland, first edition, 1986.

[7] BearShare Home page. http://www.bearshare.com/.

[8] T. Berners-Lee, R. Fielding, and L. Masinter. RFC2396: Uniform Resource Identifiers (URI): Generic syntax. http://www.ietf.org/rfc/rfc2396.txt. This document updates RFC1738 and RFC1808.

[9] D. Brickley and R. V. Guha (Eds). "Resource Description Framework (RDF) Schema Specification 1.0". W3C Recommendation, March 2000. http://www.w3.org/TR/2000/CR-rdf-schema-20000327/.

[10] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the semantic web recommendations. Technical Report HPL-2003-146, Hewlett Packard Laboratories, December 24 2003.

[11] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009, 2001.

[12] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hongang. Freenet: A distributed anonymous information storage and retrieval system in designing privacy enhancing technologie. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, Berkeley, CA, USA, July 2000. Springer-Verlag, Berlin Germany.

[13] World Wide Web Consortium. Extensible Markup Language (XML). Available at http://www.w3.org/TR/PR-xml.html, 1997.

[14] MySQL® Database Server. http://www.mysql.com/products/mysql/.

[15] FastTrack. 2001. The FastTrack Protocol. http://www.fasttrack.nu/.

[16] Freenet website. http://freenet.sourceforge.net.

[17] Gnutella website. http://gnutella.wego.com.

[18] IEEE P1484.12 Learning Object Metadata Working Group. Draft standard for learning object metadata. Technical report, IEEE learning Technology Standards Commitee (LTSC), July 2002. http://ltsc.ieee.org/wg12/files/LOM_1484_12_1_v1_Final_Draft.pdf.

[19] gtk-gnutella Home page. http://gtk-gnutella.sourceforge.net/.

[20] R.V. Guha. rdfDB : An RDF Database. Web page: http://guha.com/rdfdb/.

[21] I. Stoica and R. Morris and D. Karger and M.F. Kaashoek and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[22] KazaA Home Page. http://www.kazaa.com.

[23] Limewire Home page. http://www.limewire.com.

[24] Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can Heterogeneity Make Gnutella Scalable?

[25] Frank Manola and Eric Miller. Primer: Getting into RDF & Semantic Web using N3. http://www.w3.org/TR/REC-rdf-syntax/.

[26] Brian McBride. Jena: Implementing the RDF Model and Syntax Specification. Technical Report 20001221, Hewlett Packard Laboratories, 2000.

[27] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three implementations of SquishQL, a simple RDF query language. *Lecture Notes in Computer Science*, 2342:423, 2002.

[28] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing. Technical Report HPL-2002-57R1, Hewlett Packard Laboratories, July 14 2003.

[29] “Napster messages”. http://opennap.sourceforge.net/napster.txt.

[30] Napster website. http://www.napster.com.

[31] W. Nejdl, B. Wolf, Changtao Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch. Edutella: A P2P Networking Infrastructure Based on RDF. In *Proc. of WWW-2002*. ACM Press, 2002.

[32] Wolfgang Nejdl, Martin Wolpers, Wolf Siberski, Christoph Schmitz, Mario Schlosser, Ingo Brunkhorst, and Alexander Lser. Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. January 01 2003.

[33] N. Ntarmos and P. Triantafillou. SeAl: Managing Accesses and Data in Peer-to-Peer Data Sharing Networks. In *4th IEEE International Conference in Peer-to-Peer Computing*, August 2004.

[34] P. Suthar and J. Ozzie. The Groove Platform Architecture. Available at Groove Networks Presentation. devzone.groove.net/library/Presentations/GrooveApplicationArchitecture.ppt.

[35] S. Ratnasamy and P. Francis and M. Handley and R. Karp and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[36] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[37] Mario Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. Hypercup - hypercubes, ontologies and efficient search on p2p networks.

[38] Douglas C. Schmidt and Steve Vinoski. Object interconnections: Comparing alternative programming techniques for multi-threaded CORBA servers (column 6). January 05 1997.

[39] Andy Seaborne. RDQL – RDF Data Query Language. http://hpl.hp.com/semweb/rdql.html, 2001.

[40] Andy Seaborne. An RDF NetAPI. Technical Report HPL-2002-109, Hewlett Packard Laboratories, April 22 2002.

[41] SETI@home Home Page. http://setiathome.ssl.berkley.edu.

[42] Shareaza Home page. http://www.shareaza.com/.

[43] Stratos Idreos, Manolis Koubarakis and Christos Tryfonopoulos. P2P-DIET: Ad-hoc and Continuous Queries in Super-peer Networks. Proceedings of the IX International Conference on Extending Database Technology (EDBT04), Heraklion, Crete, Greece, March 14-18, 2004.

[44] The EdutellaProject. http://edutella.jxta.org/.

[45] Peter Triantafillou. Self Organization and Volunteering: Engineering in Very Large Scale Sharing Networks. In *SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems*, May 2004.

[46] M. Wolpers W. Nejdl, W. Siberski and C. Schmnitz. Routing and Clustering in Schema-Based Super Peer Networks. In *Proc. of IPTPS '03*, October 30, 2002.

[47] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave Reynolds. Efficient RDF storage and retrieval in jena2. Technical Report HPL-2003-266, Hewlett Packard Laboratories, January 14 2004.

[48] World Wide Web Consortium. Resource Description Framework (RDF) Model and Syntax Specification. Technical report, World Wide Web Consortium, February 1999.

[49] Beverly Yang and Hector Garcia-Molina. Designing a super-peer network. Technical Report 2002-13, Stanford University, 2002.