Summary

The present thesis explores the use of a semantic approach for indexing resources that are shared among a community of users who are scattered in a peer-to-peer network. The thesis contributes to the studies of *Semantic Indexing* approaches that can be used both with resources owned within personal memories, and with resources that are shared in a distributed network. Because the system is generic, the exact nature of the distributed community can be left undefined but our narrower focus is on applications in the e-learning domain.

To address the problem that resources have different types, an indexing system is proposed that is based on the user point of view and is performed manually. Indeed, because indices refer to the subjective information that is not necessarily contained in the resources or that are hard to extract from documents that are not textual, the indexing of such type of resources can only be done in an interactive way.

We show that a unique approach can be taken that allows one to store documents in personal or collective memories. The approach requires suitable browsing interfaces for accessing ontologies that satisfy and facilitate indexing. We therefore also define an *Indexing Patterns* system for managing ontologies that can be utilized for creating indices. The method is intended to facilitate the browsing of ontologies by showing only that part of the ontology that is useful for indexing and can be employed by users of various operating systems.

A related problem that we address concerns the difference between the *publication context* and the *retrieval context*. The solution poroposed in this thesis foresees different retrieval situations and queries during the time a resource is published and builds the index based on those assumptions.

The intended practical application encapsulates in a transparent manner the functionalities that the user requires for managing the resources. The development of a prototype is guided by *Architectural and Implementation* descriptions of the indexing system, both of which are described in the thesis.

Although the solution we offer is generic and can be used by different user communities, the approach benefits *Domain Specific Communities* in particular, by assisting specilized loose communities that are structured as peer to peer networks and that allow publishing and searching of documents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Web 2.0, often called the social Web, supports the sharing of resources and the communication between members of a loose community. The communities have various cultural goals in a specific domain and focus on the exchange of significant resources. Several fields are concerned with studying this phenomenon, one of the fundamental tasks being the development of fast access to relevant documents at one's disposal, either for individual or for collective use.

The creation of new resources is a daily concern for community members. On the one hand, users consider the resources to be private and use their personal tools for resource management. On the other hand, they willingly agree to share the resources with other people. In this scenario, a common system of resource management becomes necessary that can be used for both private and shared memories and that is at the same time very simple and not too restrictive.

In order to share and retrieve resources, it is also necessary to give them a uniform description. This is why the *file sharing* diffusion cannot be based on the document title only, as is commonly done in well-known music or video sharing systems.

One of the aims of this work is to determine whether the approach of making resources available from semantic P2P networks can be an effective solution, the main problem being the semantic indexing of resources and the attainability of the ontologies. Suitable browsing interfaces within ontologies that satisfy and facilitate the indexing will be required. Indeed, this type of indexing can only be done in an interactive way because it refers to subjective information that is not necessarily contained in resources or that is hard to extract from documents that are not necessarily textual.

1.1 Motivation

The social Web focuses on the social life of users. People are motivated to communicate, collaborate on some projects and share documents about their interests and the topics they are working on. Technical aspects as well as human, social and economic must be considered together. Our research takes place in this context, but with some peculiarities. We consider people belonging to a loose community. They do not rely on a centralized site but on a network of peer systems. They are not interested in maintaining direct and superficial contacts by only exchanging messages. They prefer to share real and useful resources that concern the focus of the community. They also want to interact with a simple and intuitive interface in order to access their privatly-owned resources and those shared with other users. The resources have to be properly described by users before becoming a part of the community. The system of resource management needs to have an index (an indexing system is required) constituted by entries that contain descriptions and access points to resources. Each member of the community owns a memory that has a private part that contains personal resources and a public part that contains those documents that have been shared with the community. A unique system for managing both parts of the memory is necessary. In a distributed system, resources are managed by a distributed index, i.e. that each member owns a part of the index in a transparent manner.

The main activities that the members of a community regularly do concern the indexing of private resources and their publication in the distributed system, on the one hand, and the retrieving of resources from the network and/or the private storage from the common requesting system, on the other hand. We identify these two activities as contexts. A context is represented by the conditions and circumstances (factors) that are relevant to the event of publication and retrieval. The publication context relates the information required to describe a resource to be published and supplied by the resource provider. The retrieval context regards the information required to search a resource and supplied by the user interested in discovering resources. The difference between a publication context and a retrieval context is one of the issues that needs to be addressed.

Documents contained in the memory may have different types (text, audio, video, archive, etc.) and different storing formats. The title of the resource is usually not sufficient for uniquely identifying a resource, as in ordinary file sharing systems [61], because the resources are created locally but their meaning is not universally known. The challenge is, therefore, (i) the development of a unique indexing system for supporting the management of resources in the public part of the user's memory and also in its private part, and (ii) the development of an application that encapsulates in a transparent manner the functionalities that the user requires for managing the resources.

Even though we mainly focus on the field of education, the exact nature of the community is not really important because the challenges that need solving, are shared by different P2P networks and the solution we propose, is generic.

Our research aims to face the problems outlined above, by showing (i) how Semantic Web standards can fully characterize indexing, and (ii) how a web based semantic desktop can benefit a community of users in sharing resources.

1.2 Proposed Solution and Contributions

Our study addresses the techniques that are needed for creating an index of managed distributed resources. We consider people who belong to a loose community where everybody takes part as a peer in a peer to peer (P2P) network. The index is distributed via a data structure called Distributed Hash Table (DHT) [24] [81] [96]. Several projects [24] [81] make use of DHTs in order to distribute the data over a large number of peers, that contribute storage to a community of users. The use of such infrastructure in general is justified by most relevant features of P2P systems, such as decentralization, scalability, security, etc. To distribute data among thousands or millions of peers involves not only an access to huge amounts of information, but also confidence in a robust system free of restriction from a central authority. Moving from the client-server paradigm to the P2P networks model, information retrieval problem becomes more articulated. Even though P2P network architectures cannot offer hyper-textual navigation between resources and lose the hierarchical organization of data, they keep some of the earlier advantages, such as the easy scalability of resources, the reliability on data transferring, and low operating costs. A P2P architecture avoids both physical and semantic bottlenecks that limit information and knowledge exchange [95].

Each node of the network contains a part of the whole data structure. The index is composed of entries that are pairs of data (*key*, *value*). In traditional file sharing systems, the key is created through the title of the resource (e.g. the title of a song). In our case, the *key* could be a set of keywords or a semantic description provided by the user for describing the resource. The *value* is the access point to the resource.

The publication is the operation of inserting a resource inside the DHT. More exactly it is the operation of inserting a new index entry in the DHT. The discovery is the operation which allows to find some resources in the network that correspond to a research key.

Many search algorithms that are based on sequences of keywords try to get more efficient results but they often lack contextual information. For improving the quality of the search and facilitating the selection of results that are more related to a specific request, semantic indexing is a valid candidate. The semantic information strictly related to a resource is included in a semantic description and then inserted

in the distributed index.

We propose a semantic approach based on domain ontologies. We mainly consider semantic indexing where semantic keys are associated with documents. This method allows reasoning based on ontologies that may enlarge the results of a search or may retrieve related documents, even complementary files. The Semantic Web languages give information a well-defined meaning, thus enabling the definition of machine processable descriptions. This remains an interesting and promising idea, especially today when the Semantic Web vision has crystallized a set of standard and well supported languages (RDF [9] , OWL [8], SPARQL [28]) and related technologies. In our solution, all the keys used for representing a document in the index represent its semantic description and are written in a language based on RDF. Ontologies used for indexing have to be included in the network by expert members and can be subsequently used by all the members. In that sense, we can consider the semantic index as a knowledge base that is created by any user who wants to share a resource.

Due to the diversity of types and formats that the resources we intend to manage may have, they would need to be manually indexed. Only some of the resources could be automatically analysed for indexing but even then, external information that cannot be found inside a document would need to be manually added in order to better describe the content and its use. Indeed, the indexing of such types of resources can only be done manually in an interactive way, because they refer to subjective information that is not contained in the resources or that is hard to extract from documents that are not text-based.

Automatic indexing of textual resources also lacks subjective information that can be provided by an expert with specific skills in a particular domain. E.g., we can extract a lot of useful information from the text of a mathematics homework but probably cannot determine automatically whether it is a difficult exercise or not. Automatic information extraction from non-textual documents, such as images, videos, etc. is possible to a certain degree but often requires sophisticated systems that are difficult to install and need a special license. This is in contradiction with our motivation. In the challenge of "bridging the semantic gap", automatically generating concept-based descriptions for multimedia information currently suffers from the lack of tools that can automatically manipulate high-level concepts that could be attached to an image or a video [23].

To resolve the issues related to the contexts of publication and retrieval, we propose to consider different retrieval situations already in the publication context by addressing possible queries to which the resource should respond positively. In doing that, we use reasoning based on the ontologies involved in the semantic description of a resource. The semantic indexing allows reasoning based on ontologies that may increase the number of results of a search.

Ontologies used for indexing would have to be presented to users in a friendly and

easy-to-use way. One of the most significant problems is the way users navigate ontologies for discovering ontological elements that are useful for indexing. They must select concepts, individuals and properties in order to annotate the resource to be indexed. Normally user interfaces provide a tree based browsing system. Tools such as Protégé [66], OntoEdit [97], OilED [7], Swoop [52] have been directed in this way. There are also several attempts to provide a graphical interface, sometimes based on 3D graph representation [17]. Most of them are meant for general purpose use and are missing a tool for selecting only useful parts of the ontologies. Furthermore, the way to display the content of the ontologies is wasteful because all the information is provided simultaneously, forcing the user to browse a large amount of data. To solve this problem we propose a navigation system within the ontologies that is able to guide the user during the selection of the important information. This system is based on patterns that we have identified during the indexing process. The solution we propose allows to associate with a document the meta-information that describes its content and to publish it in the network. The URL of the resource tied up to the meta-information allows its further access. Our solution leaves open the choice of the ontologies required for the descriptions. The user has just to select ontologies that are actually shared among communities [40], otherwise the discovery of the documents published in the network would be impossible. The only characteristic we use is the Unified Resource Identifiers (URI) of concepts, relations and instances of concepts. We also chose ontologies because we wanted any software agent to be able to reason with the knowledge base in order to build most appropriate queries when searching for resources. Indeed, we consider that the resources published in the network may be used diversely.

The existence of an ontology navigation system is not enough. Users need also tools for the access to the functionality of the community. We propose to equip the system with different *applications* as part of a *Semantic Desktop* [88][89], so that they are all integrated within a web application. Users rely on a set of tools similar to a traditional computer desktop. The back-end of the architecture consists of a set of web services managing the resources and giving access to the P2P network. A unifying web user interface gives common access to the services and allows easy communication between them.

Contributions of this thesis are the following:

- A very fine *semantic indexing* system of resources.
- *Algorithms for publishing and searching* resources compatible with the definition of publication and retrieval contexts.
- A modelling approach that supports the definition of well-structured indexing models, defined as *Patterns*. The patterns aim at supporting the creation of

user interfaces that can display only that part of the ontology that a user considers beneficial for indexing, and associate a description to a resource.

- Suitable *user interfaces* allowing the browsing of ontologies, that satisfy and facilitate the indexing of resources.
- Modelling of a *Semantic Desktop*: software architecture consisting of different *applications* that are useful for managing the semantically described resources.

1.3 Thesis Outline

We study the consequences of the context in which our research takes place and we propose an engineering system that satisfies the requirements, allowing the management of both memories in a compatible way. In particular, we describe the back-end user system organized around a service-oriented architecture and we present the design of the user interface. We show that the scalability of the system is ensured by the architecture we propose. We also analyse the indexing system that has to anticipate furture queries in order to generalize the resource descriptions.

This thesis is organized as follows:

- Chapter 2, preliminary formalisms and low level systems needed for our research.
- Chapter 3, related, state of the art work, concerning the theoretical aspects of this thesis.
- Chapter 4, conceptualization of the proposed approach.
- Chapter 5, design and implemented software.
- Chapter 6, experimental evaluation of the work.
- Chapter 7, conclusions and future work.

Chapter 2

Preliminaries

This chapter introduces the technological and scientific background required in our research. The areas of interest are *Semantic Web*, *Description Logics* and *P2P Systems*. In section 2.1 we present an overview of the *Semantic Web* and related technologies. Section 2.2 explains fundamentals on *Description Logics*. Section 2.3 outlines the concepts of *P2P Systems* and the main topologies of P2P networks.

2.1 Semantic Web

The term "Semantic Web" [10] [13] was coined by Tim Berners-Lee. His vision of the Web where information is shared, data have a format that machines can naturally understand, and where machines seamlessly collaborate among each other, has been perceived from the beginning as "*an extension of the current Web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation*". Semantic Web is about enriching the current Web with machine processable data, to enable machines to share information and thus better help humans navigate, combine and retrieve information from the vast repository of knowledge that is today's Web. Even though the Semantic Web has been widely explored during the last years, it still requires a lot of work to make the full Semantic Web dream come true. The W3C[1] has standardized a broad set of technologies that support the Semantic Web by enabling information descriptions that are formal, unambiguous and machine processable. The Semantic Web is based on a Stack (see figure 2.1)[2] of languages and protocols that are specifically designed to capture and communicate domain knowledge to diverse entities. All layers of the stack need to be implemented to achieve full visions of the Semantic Web. In this work we recount the technologies from the bottom up to OWL that are currently standardized. At

[1] http://www.w3.org/2001/sw/

[2] http://www.w3.org/2007/03/layerCake.png

the bottom there are technologies, well-known from the hypertext web, that provide the basis for the semantic web. Of these technologies, the XML markup language enables the creation of documents that are composed of structured data (Semantic web gives meaning - semantics - to structured data) and the URI provides means for uniquely identifying semantic web resources (Semantic Web needs unique identification to allow provable manipulation with resources in the top layers). Middle layers contain technologies (see section 2.1.1) standardized by W3C to enable building semantic web applications. At top layers there are technologies not yet standardized or that contain just ideas that should be implemented in order to realize the Semantic Web.

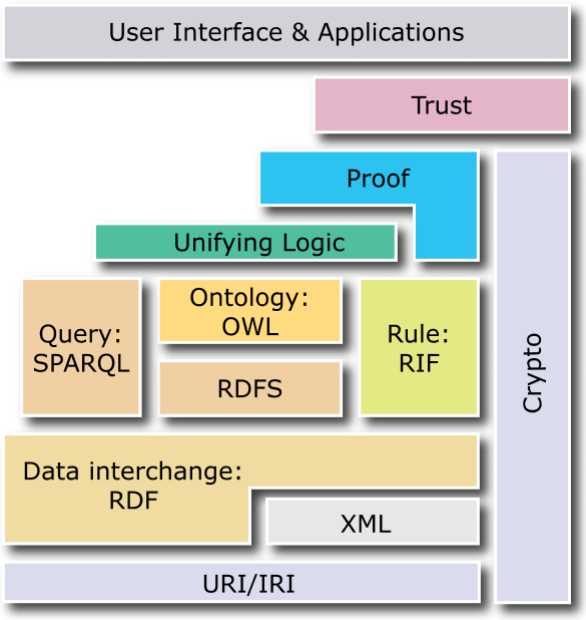


Figure 2.1. The Semantic Web stack

These languages aim to provide machine processable semantics for the domain knowledge. In the following section, we will look at the relevant representation languages that lay at the core of the Semantic Web.

2.1.1 Representation Languages

2.1.1.1 RDF

The *Resource Description Framwork* (RDF) [9] [59] [56], the basic layer of the Semantic Web stack, is the foundation for processing metadata. It provides interoperability between applications that exchange machine processable information on

the Web. Basically, RDF defines a data model for describing machine processable semantics in data. It is an assertional language and describes information in the form of *triples*. The basic data model consists of three object types: i) *Resources*, a resource may be an entire Web page; a part of a Web page; a whole collection of pages; or an object that is not directly accessible via the Web; e.g. a printed book. Resources are always named by URIs. ii) *Properties*, a property is a specific aspect, characteristic, attribute, or relation used to describe a resource. iii) *Statements*, an RDF statement consists of a specific resource, together with a named property and the value of that property for the resource in question.

These three parts of a statement are identified as *triples* and are called the *subject*, *predicate* and *object*. RDF defines triples as basic modelling primitives and introduces a standard syntax for them. An RDF document will define properties in terms of the resources to which they apply. RDF is a graph data model (see Figure 2.2): the subjects and objects of triples are represented as nodes in the graph, while the predicates are represented as directed arcs connecting pairs of nodes.

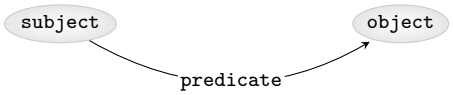


Figure 2.2. RDF graph model

A set of RDF triples is therefore referred to as an RDF *graph*, and it corresponds to the assertion of all the triples in it. Thus, the meaning of an RDF graph is the logical conjunction of all the relations asserted by its triples. A *node* in an RDF graph can be either a URI, or a literal, or a blank node. A *URI* used as a node identifies either an individual (`http://www.example.com/utc`) or a kind of thing (`http://xmlns.com/foaf/0.1/Organization`, the class representing organization in the FOAF specification [19]). A URI can be used either as the subject or the object of an RDF triple. A *literal* can appear only in the object position of a triple; it represents a value. A Literal can be *typed* when it is associated with a datatype, or *plain* in the form of a string with an optional tag specifying a language (Italian, French, English, etc.). RDF datatypes rely on the conceptual framework from XML Schema datatypes [15]. The syntactic form `"37"^^http://www.w3.org/2001/XMLSchema#integer` usually represents a typed literal, where the literal value is enclosed in quotes, and the datatype URI follows the symbol `^^`. A *blank node* is one that cannot be identified by a URI, and that is not a literal. A blank node is also referred to as anonymous node or anonymous resource, and it serves the purpose of representing resources which cannot be explicitly identified at present. Although a blank node has no name, it has a node identifier (usually having the form `_:d`, where `d` can be replaced with any string) that is unique in the scope of the RDF graph containing the blank node itself.

An *arc* in an RDF graph is always a URI, which identifies the property being asserted. For example the URI `http://xmlns.com/foaf/0.1/topic_interest` identifies a property that relates the organization topic of interest.

RDF has model-theoretic semantics, which is specified in [47], and an XML serialization, which is defined in [9]. An RDF graph can be serialized to different XML documents which have the same interpretation. Although the XML format has some advantages, the RDF/XML serialization is very verbose, and hardly readable for human beings. URIs are written using qualified names: for example, `http://xmlns.com/foaf/0.1/Organization` is shortened to `foaf:Organization`, provided that the `foaf` prefix has been properly declared. Note that, for the sake of brevity we will usually omit the declaration of namespaces prefixes. We will also adopt the traditional convention of graphically representing URIs and blank nodes as ellipses, and literals as rectangles [59].

Figure 2.3 shows an example of RDF graph. First it defines the prefixes of qualified names used to shorten the URIs. The figure describes a resource ex:utc whose type is foaf:Organization. The resource has a topic 'interest' which is a ex:Field_of_interest labelled as a xsd:String "Computer Science".

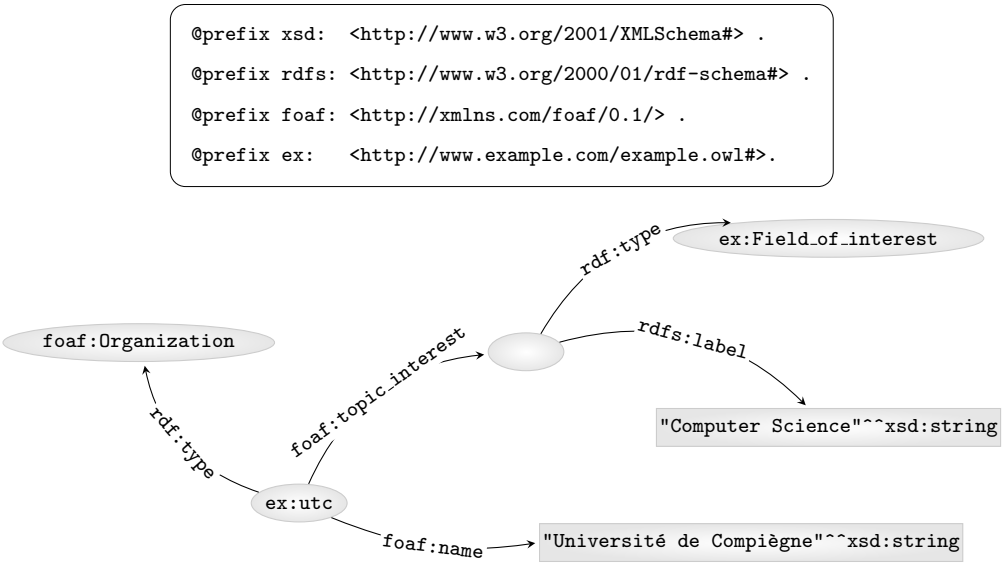


Figure 2.3. RDF graph example

2.1.1.2 RDFS

Even though RDF is a simple and flexible language, the modeling primitives it offers are very basic and the language has limited expressiveness. Therefore, the Semantic Web stack layers more powerful languages on top of RDF. Those languages provide

more expressive constructs but they retain the basic RDF characteristics, and ultimately yield RDF graphs. The first of such layers is RDF Schema (RDFS) [18], which defines a vocabulary of RDF resources that can be used to describe properties of other RDF resources. RDF Schema extends (or enriches) RDF by assigning an externally specified semantics to specific resources, e.g., to *rdfs:subClassOf*, to *rdfs:Class* etc. RDFS does not actually impose any schema on RDF; rather it augments RDF providing information about the interpretation of RDF graphs.

Core classes are *rdfs:Resource*, *rdf:Property*, and *rdfs:Class*. Everything that is described by RDF expressions is viewed to be an instance of the class rdfs:Resource. The class rdf:Property is the class of all properties used to characterize instances of rdfs:Resource, i.e., each relation is an instance of rdf:Property. Finally, rdfs:Class is used to define concepts in RDFS, i.e., each concept must be an instance of rdfs:Class.

Core properties are *rdf:type*, *rdfs:subClassOf*, and *rdfs:subPropertyOf*. The rdf:type relation models *instance-of* relationships between resources and classes. A resource may be an instance of more than one class. The rdfs:subClassOf relation models the subsumption hierarchy between classes and is supposed to be transitive. Again, a class may be a subclass of several other classes. The rdfs:subPropertyOf relation models the subsumption hierarchy between properties. If some property P_2 is a rdfs:subPropertyOf another property P_1, and if a resource R has a P_2 property with a value V , this implies that the resource R also has a P_1 property with a value V.

Core constraints are *rdfs:range* and *rdfs:domain*, which can be used to couple properties with value and subject classes in a global way. Multiple domain/range constraints on single properties are interpreted through conjunctive semantics: if a property P has as its domain classes A and B, an instance a that is the subject of a statement using P , is entailed to be an instance of both A and B.

RDFS can be regarded as a simple Description Logic with limited expressiveness, and it is actually the language underpinning the actual Web Ontology Language OWL [8].

2.1.1.3 OWL

The Semantic Web uses XML based formalism to define customized tagging schemes and RDF flexible approach to representing data. Following the Stack, the next element required for the Semantic Web is a web ontology language which can formally describe the semantics of classes and properties used in web documents. The term *Ontology* is used in different ways by different people. Pidcock [73] writes that "People use the word to mean different things, e.g.: glossaries and data dictionaries, thesauri and taxonomies, schema and data models, and formal ontologies and inference." Uschold [100] states "An ontology may take a variety of forms, but necessarily it will include a vocabulary of terms, and some specification of their meaning. This

includes definitions and an indication of how concepts are interrelated which collectively impose a structure on the domain and constrain the possible interpretations of terms." The word ontology in philosophy, since its origin denotes the study of the nature of existence, as well as of the basic categories of being and their relations. Gruber [41] defines an ontology as "a specification of a conceptualization"[3]. Thus we can regard OWL as a language for writing specification of conceptualizations. An ontology defines the terms used to describe and represent an area of knowledge. People, databases, and applications use ontologies to share domain information (a domain is just a specific subject area or area of knowledge, like medicine, tool manufacturing, real estate, automobile repair, financial management, etc.). Ontologies include computer-usable definitions of basic concepts in the domain and the relationships among them. They encode knowledge in a domain and also knowledge that spans domains. In this way, they make that knowledge reusable.

The word ontology has been used to describe knowledge bases with different degrees of structure. These range from simple taxonomies, to metadata schemes, to logical theories. The Semantic Web needs ontologies with a significant degree of structure, specifying descriptions for the following kinds of concepts:

- *classes* (general things) in the many domains of interest;
- the *relationships* that can exist among things;
- the *properties* (or *attributes*) those things may have.

Ontologies are usually expressed in a logic-based language, so that detailed, accurate, consistent, sound, and meaningful distinctions can be made among the classes, properties, and relations. Some ontology tools can perform automated reasoning using the ontologies, and thus provide advanced services to intelligent applications such as: conceptual/semantic search and retrieval, software agents, decision support, speech and natural language understanding, knowledge management, intelligent databases, and electronic commerce.

The core constructs of OWL derive from a long history of languages based on Description Logics, including DAML+OIL[4], OIL [50], and CLASSIC [16]. OWL is actually a family of three increasingly expressive sublanguages designed for the use by specific communities of implementers and users:

- *OWL Lite*, supports classification hierarchies, and simple constraints, such as number restrictions, but only with values zero and one;

[3]http://www-ksl.stanford.edu/kst/what-is-an-ontology.html
[4]DAML+OIL, March 2001, http://www.daml.org/language/

- *OWL DL*, is a compromise between expressiveness and computational complexity; OWL DL corresponds to the Description Logic, and it guarantees completeness (all entailments are computed), and decidability (all computations will finish in a finite time);

- *OWL Full*, gives the maximum expressiveness in the detriment of computational efficiency (incompleteness); for example, OWL Full treats classes simultaneously as collections of individuals and as individuals in their own right.

Figure 2.4 shows an OWL fragment; it highlights some differences in the expressiveness among the three OWL sublanguages: for example OWL DL enables the definition of disjoint classes, and OWL-Full allows one to declare that two classes are the same (in this case the classes are treated as individuals).

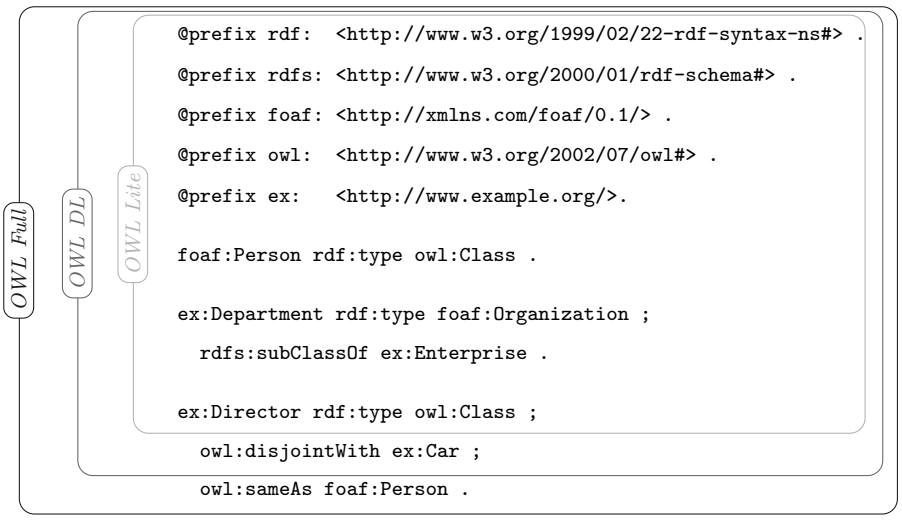


Figure 2.4. OWL example

We observe that every OWL sublanguage can be mapped to RDF, as described in [70]. In this thesis we will work mainly at the RDF level; and when we use ontologies, we will usually refer to OWL DL ontologies (unless otherwise specified).

2.1.2 SPARQL for Querying Data

SPARQL is the query language for RDF that allows to query for triples from an RDF triple store. It has been standardized by W3C [75]. Superficially it resembles the Structured Query Language (SQL) [6] used to get data from a relational database. In many respects, though, a triple data source and a relational database are fundamentally different. A relational database is table-based, meaning that data is stored

in fixed tables with a foreign key relationship that defines the relationship between rows in the tables. A triple data source stores only triples. Triples are added continuously while describing a thing. Relational databases constrain the model to the layout of the database. RDF doesn't use foreign and primary keys either. It uses URIs, the standard reference format for the Semantic Web. By using URIs, a triple data source immediately has the potential to link to any other data in any triple data source. That plays to the distributed strengths of the Web. Because triple data source are large collections of triples, SPARQL expresses queries by defining a template for matching triples, called a Graph Pattern. The triples in a triple data source make up a directed labelled graph that describes a set of resources. To get data out of the triple data sources using SPARQL, it is necessary to define a pattern that matches the statements in the graph.

SPARQL has capabilities for querying required and optional graph patterns and their conjunctions and/or disjunctions; it also supports value testing and filtering of results. SPARQL queries can yield either result sets or RDF graphs.

A SPARQL *graph pattern* is made of *triple patterns*. A triple pattern is essentially an extension of an RDF triple allowing *variables* in the subject, predicate, and object positions. The SPARQL triple pattern `{ ex:utc foaf:topic_interest ?y . }` would match any RDF triple whose subject is the resource `ex:utc` and whose predicate is given by the URI `foaf:topic_interest`.

SPARQL supports various kinds of graph patterns: the basic one is a set of triple patterns, and each query result must match all triple patterns in the set. Graph patterns can be combined in groups (where each query result must match all graph patterns) or unions (where any graph pattern in the union can match). Additionally, a graph pattern can contain an optional subgraph, which extends the results of the non-optional subgraph without causing the overall graph pattern to fail. Finally, SPARQL allows for filtering query results with built-in or custom functions, and for modifying solution sequences (ordering, uniqueness, etc.).

SPARQL has various *query forms*, among which `SELECT` (returning the variable bindings that result from each match of the query graph pattern), `ASK` (returning a boolean that indicates whether the query graph pattern matches or not), and `CONSTRUCT` (returning an RDF graph specified by a graph template).

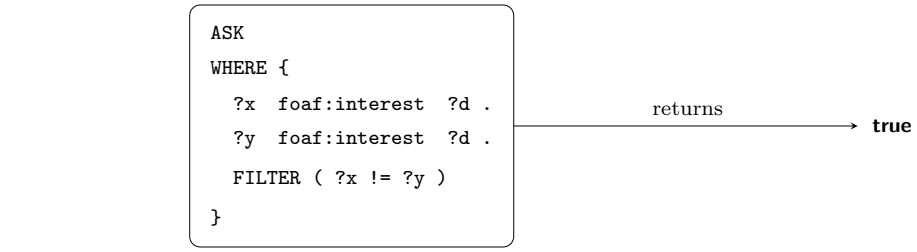
Figure 2.5 shows some examples of SPARQL queries. The related graph pattern requires two different resources identified by the variables `?x` and `?y`, and interested in the same resource `?d`.

The `ASK` query (see figure 2.5(a)) returns the boolean value `true`.

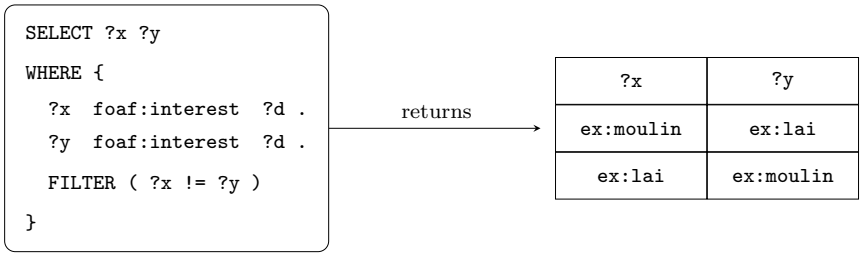
The `SELECT` query (see figure 2.5(b)) returns a list of solutions, where every solution has a mapping with the required variables `?x` and `?y`.

The `CONSTRUCT` query (see figure 2.5(c)) builds an RDF graph, where the couples of people who share an interest in the same resource are members of the same organization. In this case, two groups are generated and the graph pattern returns

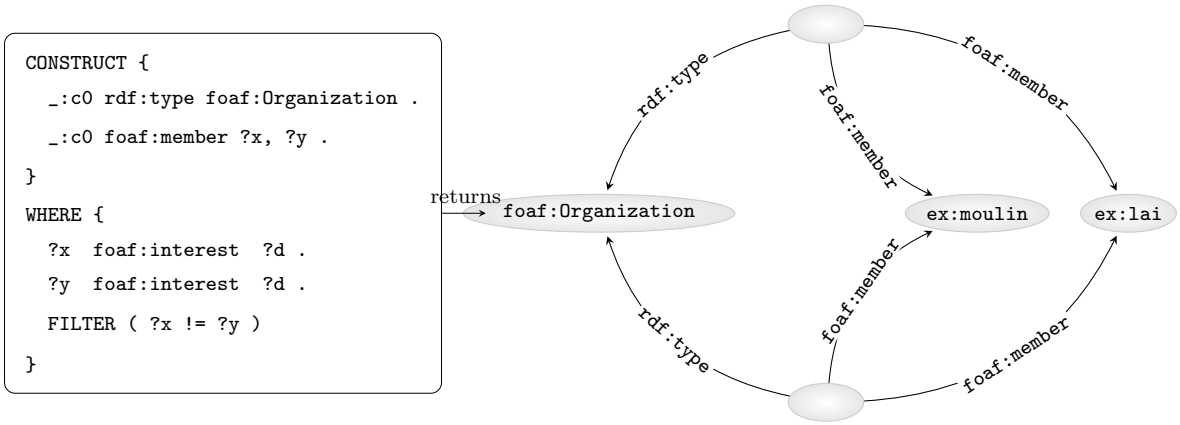
two solutions mappings.



(a) *ASK* query form



(b) *SELECT* query form



(c) *CONSTRUCT* query form

Figure 2.5. SPARQL examples

2.1.3 RDF and SPARQL Syntax

In this section we present a formal description of the syntax of the key elements of RDF and SPARQL. A comprehensive description of the full syntax and semantics of these languages, which are specified by W3C[5] can be found in [56, 9] and [47] for RDF syntax and semantics respectively, and in [75] for SPARQL syntax and semantics.

The syntax of RDF and SPARQL is based on the following sets of symbols:

Definition 2.1 (*Symbols*)**.** The following are infinite sets, pairwise disjoint:

- I : the set of *IRIs*
- B : the set of *blank nodes*
- L : the set of RDF *literals*
- V : the set of *variables*

Additionally, the set $T = I \cup B \cup L$ is the set of RDF *terms*. ⌟

The elements of the set I are *Internationalized Resource Identifiers* as defined in [33]. The blank nodes of the set B are also referred to as *anonymous resources* or *bnodes* [56]; a blank node is neither an IRI, nor a literal, nor a variable; it has a Node ID which is limited in scope to a serialization of a particular RDF Graph (see definition 2.3). A blank node serves the purpose of representing resources which cannot be named at present, and it is essentially equivalent to an existentially quantified variable. Finally, the elements of the set L identify values such as numbers and dates by means of lexical representations [56]. The set L is divided into two disjoint subsets, containing respectively *plain literals* (strings combined with an optional language tag, which are interpreted as plain text in a natural language) and *typed literals* (strings combined with a datatype URI, interpreted as values of that datatype).

The *triple* is the basic syntactic element of RDF. It corresponds to a simple statement. A set of triples gives an RDF graph.

Definition 2.2 (*RDF Triple*)**.** An RDF triple is a tuple

$$(s,p,o) \in (I \cup B) \times I \times (I \cup B \cup L)$$

where s is the *subject*, p is the *predicate*, and o is the *object*. A *ground RDF triple* is one with no blank nodes. ⌟

[5] World Wide Web Consortium, http://www.w3.org/

An RDF triple has a graphical representation: s is represented by an ellipse, o is represented by either an ellipse (when $o \in I \cup B$) or a rectangle (when $o \in L$), and p is represented by an arrow from s to o. Each graphical element (ellipse, rectangle, and arrow) is labelled with the corresponding value of the tuple.

Definition 2.3 (*RDF Graph*)**.** An RDF graph G is a set of RDF triples:

$$G \subset (I \cup B) \times I \times (I \cup B \cup L)$$

Usually, the $term(G)$ denotes the set of terms of G, that is the set of elements of T that appear in G, and $blank(G)$ denotes the set of blank nodes of G, that is $blank(G) = term(G) \cap B$. A *ground RDF graph* is one with $blank(G) = \emptyset$. Two RDF graphs are *equivalent* if they differ only in the identifiers of their blank nodes. ⌟

RDF graph equivalence is formally defined in [56, Graph Equivalence][6]. The graphical representation of the RDF triples of G is a directed graph as described in section 2.1.1.1.

Definition 2.4 (*RDF dataset*)**.** An RDF dataset is a set:

$$D = \{G_0,(u_1,G_1),\ldots,(u_n,G_n)\}$$

where $n > 0$, $G_0,\ldots,G_n$ are RDF graphs, $u_1,\ldots,u_n$ are distinct IRIs ($u_i \neq u_j$ for all $i \neq j$). G_0 is referred to as *default graph*, and (u_i,G_i) is referred to as *named graph*. ⌟

Definition 2.5 (*Mapping*)**.** A mapping (or *solution mapping*) is a partial function

$$\mu : V \longrightarrow T$$

whose domain $dom(\mu) \subseteq V$ is the set of variables where μ is defined, and T is the set of RDF terms (see definition 2.1). ⌟

Definition 2.6 (*Filter constraint*)**.** Let T be the set of RDF terms (as defined in 2.1). A filter constraint is inductively defined as follows:

- the function $r : 2^T \longrightarrow \{true,false\}$ is a filter constraint: it takes a set of RDF terms and returns a boolean value;
- if r is a filter constraint, then $\neg r$ is a filter constraint;
- if r_1,r_2 are filter constraints, then $r_1 \wedge r_2$ is a filter constraint;
- if r_1,r_2 are filter constraints, then $r_1 \vee r_2$ is a filter constraint.

[6] http://www.w3.org/TR/rdf-concepts/#section-graph-equality

Given a mapping μ and a filter constraint r, the expression $r(\mu)$ indicates the evaluation of the filter constraint r over the mapping specified by μ. ⌟

The simplest syntactic element of SPARQL is the triple pattern, which intuitively corresponds to an RDF triple allowing variables in the subject/predicate/object positions:

Definition 2.7 (*SPARQL Triple Pattern*)**.** A SPARQL triple pattern is a tuple

$$t = (s',p',o') \in (T \cup V) \times (I \cup V) \times (T \cup V)$$

A *ground triple pattern* is a triple with no variables. Given a SPARQL triple pattern t, $var(t)$ usually denotes the set of variables appearing in t. ⌟

Note that there are two relevant differences between a SPARQL triple pattern (s',p',o') and an RDF triple (s,p,o) (see definition 2.2): (i) s' can be an RDF literal whereas s cannot, and (ii) both s', p', and o' can be variables whereas s, p and o cannot.

The combinations of triple patterns yield more complex syntactic forms, which are named SPARQL graph patterns:

Definition 2.8 (*SPARQL graph pattern*)**.** A SPARQL graph pattern is inductively defined as follows:

- a triple pattern t is a graph pattern;
- if γ_1,γ_2 are graph patterns, then $(\gamma_1\ AND\ \gamma_2)$ is a graph pattern;
- if γ_1,γ_2 are graph patterns, then $(\gamma_1\ UNION\ \gamma_2)$ is a graph pattern;
- if γ_1,γ_2 are graph patterns, then $(\gamma_1\ OPTIONAL\ \gamma_2)$ is a graph pattern;
- if γ is a graph pattern and r is a filter constraint (see definition 2.6) with $var(r) \subseteq var(\gamma)$, then $(\gamma\ FILTER\ r)$ is a graph pattern.

Given a SPARQL graph pattern γ, $var(\gamma)$ usually denotes the set of variables that appear in γ. ⌟

Definition 2.8 follows the algebraic approach of [71], which has been further refined in [72, 1]. Note that there are some differences between definition 2.8 and the W3C recommendation [75]. Firstly, the W3C recommendation does not use the keyword AND but concatenates triple patterns and graph patterns enclosing them in curly brackets: in case of triple patterns, the expression $t_1\ AND\ t_2$ is written $\{t_1\ t_2\}$; in case of generic graph patterns, the expression $\gamma_1\ AND\ \gamma_2$ is written $\{\{\gamma_1\}\{\gamma_2\}\}$ (We will use both syntactic forms interchangeably). Secondly, the W3C

recommendation does not explicitly impose any restriction on the variables of filter constraints used in graph patterns of the form $(\gamma\ FILTER\ r)$, whereas definition 2.8 requires that $var(r) \subseteq var(\gamma)$, that is r is a *safe filter*. This requirement corresponds to a more general intuition of the meaning of filter constraints, and does not include any limitations. In fact, [1] shows that non-safe filters in SPARQL are superfluous. Thirdly, the W3C recommendation allows graph patterns with the following syntax: $(GRAPH\ x\ \gamma)$ where $x \in I \cup V$ is either an IRI or a variable, and γ is a graph pattern. Such graphs are referred to as *named graphs*, and essentially enable selective queries on RDF graphs from an RDF dataset (see definition 2.4). We omitted named graphs from definition 2.8 because $\mathcal{G}_p\mathcal{DL}$-formulae do not use them. Finally, the W3C recommendation designates graph patterns according to table 2.1.

Table 2.1. Graph patterns designation

Designation	**Graph pattern** (t_i is a triple pattern, and γ_i is a graph pattern)
Basic Graph Pattern or *Template Pattern*	$t_1\ AND\ t_2\ \ldots\ AND\ t_n$
Group Graph Pattern	$\gamma_1\ AND\ \gamma_2\ \ldots\ AND\ \gamma_n$
Alternative Graph Pattern	$\gamma_1\ UNION\ \gamma_2$
Optional Graph Pattern	$\gamma_1\ OPTIONAL\ \gamma_2$

The *basic graph pattern* (or template pattern) can also be regarded as a set of triple patterns $\beta \subset (T \cup V) \times (I \cup V) \times (T \cup V)$, which is more evident when using the notation with brackets: $\{t_1,t_2,\ldots,t_n\}$. The basic graph pattern corresponding to the empty set is also referred to as the *empty graph pattern*. The name *template pattern* is usually adopted in the definition of a particular SPARQL query form named $CONSTRUCT$, which allows the construction of an RDF graph based on a *template*. In general, queries are defined as follows:

Definition 2.9 (*SPARQL query*)**.** A SPARQL query is given by a tuple

$$(\gamma, D, Q)$$

where γ is a SPARQL graph pattern (see definition 2.8), D is an RDF dataset (see definition 2.4), and Q is one of the following query forms:

- ASK
- $SELECT\ X$

- $CONSTRUCT\ \tau$

where $X \subset V$ is a set of variables such that $x \in X \rightarrow x \in var(\gamma)$, and τ is a SPARQL template pattern (see table 2.1). ⌟

Note that the W3C recommendation [75] defines a SPARQL query in terms of a SPARQL algebra, an abstract intermediate language for the expression and analysis of queries. An early description of SPARQL algebra can be found in [28]. The W3C recommendation describes how to convert SPARQL graph patterns into SPARQL algebra expressions. The translation is performed by the SPARQL interpreter when evaluating a query.

2.1.4 Reasoning

A reasoner is a piece of software able to infer logical consequences from a set of axioms or asserted facts. In practice, a reasoner makes inferences either about classes constituting an ontology or about individuals constituting a knowledge base. Ontology classification arranges classes defined by logical expressions into a hierarchy. This reasoning task is normally related to ontology development. Our approach concerns the query answering with respect to ontology based information retrieval. The Semantic Web requires high-performance storage and reasoning infrastructure in order to match the demand of indexing structured data with the use of ontologies. The major challenge with building such infrastructures is the expressivity of the underlying standards such as RDF(s) and OWL [54].

For the first case of reasoning (concept classification), we may use different available engines related to ontology languages of the Semantic Web like OWL and RDF(s) [51]. Among the most popular are RacerPro, FaCT++ and Pellet. Pellet, in particular, is an OWL DL reasoner based on the tableaux algorithm [5] developed for expressive Description Logics. Pellet parses OWL documents into triples and separates them into TBox (axioms about classes), ABox (assertions about individuals) and RBox (axioms about properties), which are passed to the tableaux based engine. Logic relations contained into the ontology and constituting classes, individuals, properties allows to create new axioms.

An interesting feature of Pellet is its usability for ontology analysis and repair. As explained in [67] OWL has two major dialects, OWL DL and OWL Full, with OWL DL being a subset of OWL Full. All OWL knowledge bases are encoded as RDF/XML graphs. OWL DL imposes a number of restrictions on RDF graphs, some of which are substantial (e.g., that the set of class names and individual names be disjoint) and some less so (that every item have a type triple). Ensuring that an RDF/XML document meets all the restrictions is a relatively difficult task for authors, and many existing OWL documents are nominally OWL Full, even though their authors intended for them to be OWL DL. Pellet incorporates a number of

heuristics to detect DLizable OWL Full documents and repair them, i.e. making them compliant with DL characteristics.

The second case of reasoning based on the structure of the ontology applies the semantics rules of OWL to a knowledge base. This adds new assertions to the knowledge base (the first case adds new axioms to the ontology). In [54] the authors explain two principle strategies for rule-based inference, forward-chaining and backward-chaining. Their approach is based on inferred closure and known as materialization. Through the inferred closure, a knowledge base is extended with all the facts inferred by the application of semantic rules.

2.2 Fundamentals on Description Logics

The languages of the Semantic Web are reinforced by logic formalisms. In this section we will see some fundamentals of them. A logic is a formal language that defines the syntax of the so called well-formed formulae (the set of statements that one can make in that particular logic) and has a model theory, which unambiguously specify the semantics of the well-formed formulae. Logic studies in general the principles of valid inference and correct reasoning: a logic is not concerned with establishing the truth or falsity of a statement, but only with consistently deducing the truth of a statement from that of the others.

Description Logics (DL) are a family of knowledge representation formalisms that allow for the construction of symbolic representations of an application domain by formally defining individual objects, classes of objects with similar characteristics, and relationships among them. Classes are usually arranged in class hierarchies based on a subclass/superclass relationship (partial ordering). The use of Description Logics in knowledge representation systems can be traced back to the KL-ONE [90] and CLASSIC [16]. See [3] for a comprehensive presentation of Description Logics.

An application domain is formally defined by DL; the domain is given through its *terminology*, which consists of a finite set of classes and roles (i.e. relations). The terminology is usually referred to as *TBox*, and it represents the intentional knowledge of the application domain. The TBox may also contain a finite set of *axioms*, which specify additional concepts, properties of roles, or relations between concepts. Individual objects are specified by *assertions* based on the definitions given in the TBox. The set of assertions is usually referred to as *ABox*, and it defines the extensional knowledge of the application domain. A TBox is usually considered stable and does not change very often, whereas an ABox is usually considered contingent, and therefore mutable. A TBox and a corresponding ABox are usually jointly referred to as a *knowledge base*.

A set of *constructors* supplies the DL languages. *Constructors* combine the basic classes and roles of the terminology into complex ones. A typical Description Logic Knowledge Representation System contains a knowledge base, which is based on a specific Description Logic language, and a set of *reasoning services*, which are useful in various information processing applications, and which correspond to the way humans understand and reason about the world. The objective of the reasoning services is to ensure *soundness* (only valid relationships are identified), *completeness* (all valid relationships are identified), and *tractability* (relationships can be computed in a *reasonable* time). The set of constructors and the various kinds of axioms determine the differences in the various Description Logic languages, and affect not only their expressive power, but also the decidability and complexity of the reasoning services.

2.2.1 Syntax and Semantics

The syntax of Description Logics languages is based on three pairwise disjoint sets: the set of *atomic concepts* $\mathcal{S}_{AC}$, the set of *atomic roles* $\mathcal{S}_{AR}$, and the set of *individuals* $\mathcal{S}_{Ind}$. The symbols $\top$ (*top*) and $\bot$ (*bottom*) represent the most general and the least general concept, respectively. Concepts and roles built using constructors supported by a DL language are called *complex*.

Example 2.1 (Concepts, roles, and individuals)**.** Let's consider two atomic concepts, `Person` and `Organization`, and an atomic role `member`. `Person(Moulin)` is an individual of type `Person`, and `Organization(UTC)` is an individual of type `Organization`. The complex concept `Professor` can be defined as a `Person` related to a `Organization` through the `member` role.

The URI `http://xmlns.com/foaf/0.1/Person` represents the concept `Person` in the FOAF specification [19]. If the URI `http://www.example.com/ClaudeMoulin` identifies Claude Moulin, then the RDF triple

```
ex:ClaudeMoulin rdf:type foaf:Person .
```

is equivalent to the Description Logic expression`Person(ClaudeMoulin)`, asserting that `ClaudeMoulin` is an individual belonging to the set identified by the atomic concept `Person`. (Note that the `rdf` and `foaf` prefixes identify respectively the RDF and FOAF vocabularies, and the `ex` prefix identifies the fictitious namespace `http://www.example.com/`.) ⌟

The semantics of a Description Logic language is given in terms of an *interpretation* $\mathcal{I}$.

Definition 2.10 (Interpretation)**.** An interpretation $\mathcal{I}$ is defined as a tuple

$$\mathcal{I} = (\Delta^{\mathcal{I}}, (\cdot)^{\mathcal{I}})$$

where $\Delta^{\mathcal{I}}$ is a non-empty set called the *domain of the interpretation*, and $(\cdot)^{\mathcal{I}}$ is the *interpretation function* [4]. ⌟

The interpretation function $(\cdot)^{\mathcal{I}}$ is actually specified as a set of three functions (in the following 2^A denotes the powerset of the set A):

- $(\cdot)^{\mathcal{I}}_{AC} : \mathcal{S}_{AC} \longrightarrow 2^{\Delta^{\mathcal{I}}}$ is a function mapping atomic concepts to subsets of the interpretation domain: $\forall C \in \mathcal{S}_{AC}$, $(C)^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$;
- $(\cdot)^{\mathcal{I}}_{AR} : \mathcal{S}_{AR} \longrightarrow 2^{\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}}$ is a function mapping atomic roles to subsets of binary relations over the interpretation domain: $\forall R \in \mathcal{S}_{AR}$, $(R)^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$;
- $(\cdot)^{\mathcal{I}}_{Ind} : \mathcal{S}_{Ind} \longrightarrow \Delta^{\mathcal{I}}$ is a function mapping individuals to elements of the interpretation domain: $\forall i \in \mathcal{S}_{Ind}$, $(i)^{\mathcal{I}} \in \Delta^{\mathcal{I}}$.

The interpretation function is conveniently extended to the sets of complex concepts and roles to define the semantics of the constructors allowed in the various DL languages.

The TBox of a DL language contains a finite set of axioms which can introduce new concepts and roles, assert subclass/superclass relationships, and assert properties of roles (such as transitivity).

2.3 P2P Systems

P2P systems appeared on the Internet to support applications that harness the resources of a large number of autonomous participants (called peers). In many cases, these peers form self-organizing networks that are layered on top of conventional Internet protocols and have no centralized structure. P2P systems are usually characterized by the absence of a centralized authority that drives the system's components, and base their functioning on the self-organization of the entities that take part in the system by playing a symmetrical role. For these reasons, the applications best suited for P2P implementation are those where centralization is not possible, relations are transient, and resources are highly distributed [74]. Another key aspect of P2P systems is the ability to provide inexpensive, but at the same time scalable, fault tolerant and robust computing infrastructures. File sharing services like Gnutella, are distributed systems where the contribution of many participants with small amounts of disk space results in a very large distributed database. In the SETI@home project [7] users volunteer their CPU resources, making up a large-scale signal processing infrastructure to support the research of extraterrestrial life.

P2P and classical distributed computing are both concerned with enabling resource sharing within distributed communities. However, different base assumptions have led to distinct requirements and technical directions [36]. P2P systems have focused on resource sharing in environments characterized by potentially millions of users, most with homogeneous desktop systems and low-bandwidth, intermittent connections to the Internet. As such, the emphasis has been on global fault-tolerance and massive scalability. In contrast, classical distributed systems have arisen from collaboration between generally smaller, better-connected groups of users with different resources to share.

Despite these differences, the long-term evolution of classical distributed computing and P2P seems likely to converge at least in some regards, as distributed systems expand in scale and incorporate more transient services and resources, and as P2P researchers consider a broader class of applications [36].

2.3.1 P2P history

P2P networking has divided research cycles. The traditional distributed computing community views these young technologies as "upstarts with little regard for, or memory of, the past"; evidence supports this view in some cases. Others welcome an opportunity to revisit past results, and to gain practical experience with large-scale distributed algorithms. An early use of the term "P2P computing" is in IBM's Systems Network Architecture documents over 25 years ago, but publicly came to

[7] Seti@Home Project. http://setiathome.ssl.berkeley.edu

fore with the rise and fall of Napster[8] file sharing application in 1999.

2.3.1.1 P2P vs Client-Server Model

P2P systems can be contrasted with asymmetric client/server systems, in which a server (usually a more powerful and better connected machine) runs for long periods of time and delivers storage and computational resources to some number of clients. As a side effect, the server becomes a performance and reliability bottleneck and the undesirable features need to be mitigated with techniques such as replication, load balancing, or request routing, and significant investments in high-end machines, high-bandwidth connectivity, rack space and so forth. All of that, suggests that the support of a centralized solution is a viable option provided in an economic incentive or a business model that justifies capital and administrative expenses.

A natural evolution of this thinking is to include the clients' resources in the system, an approach that becomes increasingly attractive as the performance gap between desktop and server machines narrows, and broadband networks dramatically improve client connectivity. Thus, P2P systems evolve from client/server systems by removing the asymmetry in roles: clients are also servers that allow access to their resources, actively participating in the service supply. Work (be it computation, or file sharing) is partitioned between all peers, so that a peer consumes its own resources on behalf of others (acting as a server), while asking other peers to do the same for its own benefit (acting as a client). As in the real world, this cooperative model may break down if peers are not provided with incentives to participate, which in successful stories like Napster or Gnutella [55] turned out to be just the nature of the content being shared.

Another viewpoint from which one can describe these systems is the use of intermediaries. The Web (and client/server file systems such as NFS and AFS) uses caches to reduce average latency and networking load, but these caches are typically arranged statically. P2P systems partition work dynamically among cooperative peers to achieve locality oriented load balancing. Content distribution systems such as PAST [83] and Pasta [62] use demand-driven strategies to distribute data to peers close in the network to that demand.

The classical distributed systems community would claim that many of these ideas were present in the early work on fault tolerant systems in the 1970s. For example the Xerox Network System's name service, Grapevine [91], included many traits mentioned here. Other systems that can be construed as P2P systems include Net News (NNTP is certainly not client-server) and the Web's Inter-cache protocol, ICP. The Domain Name System also includes zone transfers and other mechanisms that are not part of its normal client/server resolver behaviour.

[8] Napster. Napster media sharing system. http://www.napster.com/.

2.3.1.2 First generation: Napster

The Napster file sharing system started in 1999 allowing users to "share" audio files stored on their own hard drives. In Napster (see figure 2.6), peers stored locally their collection of files, while Napster ran a central server storing only the index of files available within the peer community. To retrieve a desired song, users issued keyword-based search requests to this central server and obtained the IP address of peers storing matching files. The user could then download the desired files directly from one of these peers.

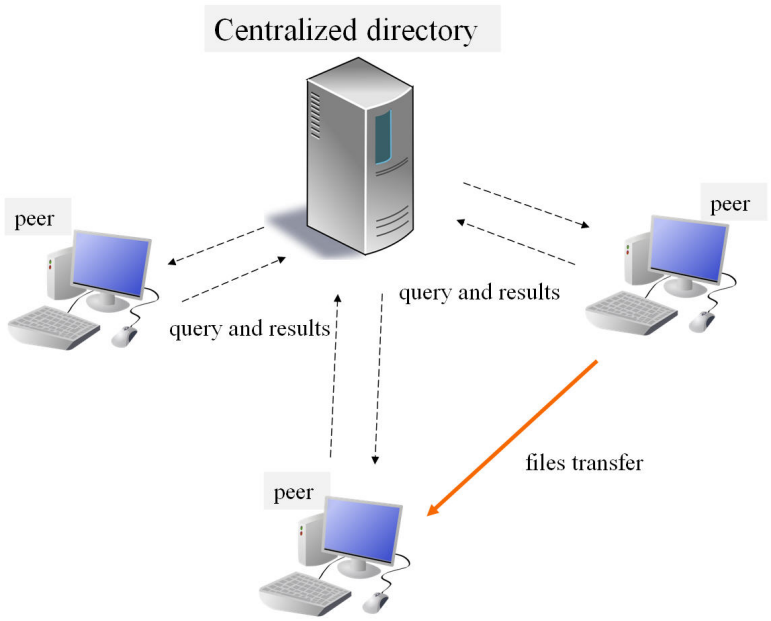


Figure 2.6. Napster, centralized architecture

Clearly, Napster did not display a "pure" P2P architecture, as only the content storage and exchange were distributed among the peers, while file indexing and lookup was on a central location administered by Napster (the company). However, because it dramatically simplified the task of obtaining music on the Internet, Napster became popular, reaching nearly 50 Million users within the first year of service. Over time, Napster's centralized directory became both a severe bottleneck and a single point of failure for legal, economic, and political attacks. Napster was eventually shut down by court order for helping users infringe copyright. Napster's success was attributable to online music sharing being a "killer application". Moreover, it demonstrated the potential in harnessing client resources to satisfy their need for a service. With the demise of Napster, there arose a desire within the music-sharing community for a fully decentralized service that would not be susceptible to a similar

legal attack. The projects that rose to the challenge stimulated important technical developments in distributed object location and routing, distributed searching, and content dissemination.

2.3.1.3 Second generation: Gnutella, Freenet, Kazaa

Gnutella is a distributed search protocol adopted by several file-sharing applications, which dispensed with the centralized directory and distributes also the file indexes and lookup. Gnutella peers locate content sources flooding their neighbourhood with search queries messages (see figure 2.7). Despite measures to limit and restrict flooding, several studies and user experience found that sometimes the volume of queries and control of traffic caused excessive network load, decreasing the chance of satisfying a given query, as well as the amount of bandwidth left for actual file transfers.

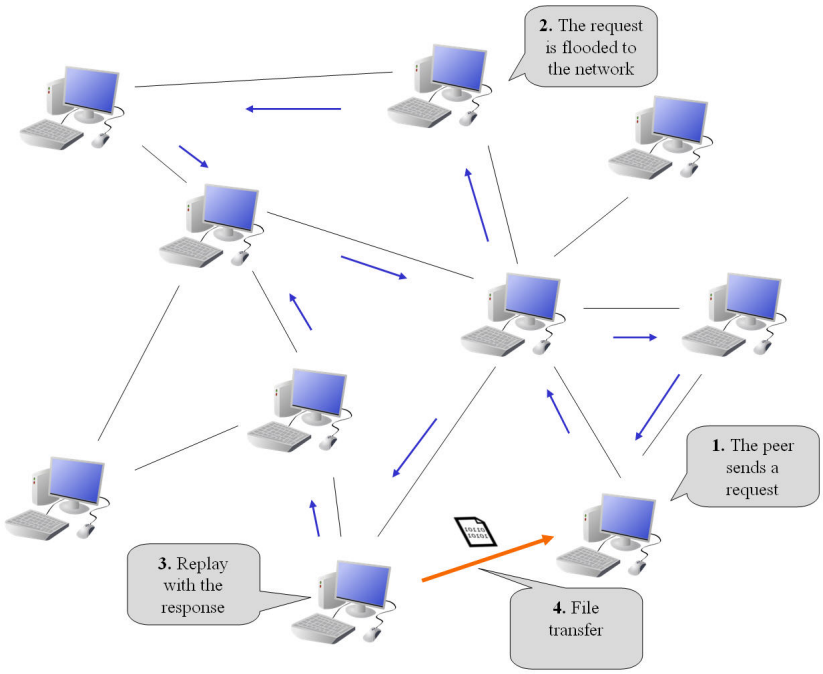


Figure 2.7. Gnutella, fully decentralized architecture

Other systems for content location, including Freenet [26], added mechanisms to route requests to the node where the content is likely to be, in a best effort partitioning of the networks' content. Systems for file sharing such as Kazaa [57], as well as recent Gnutella evolutions, added structure to P2P file-sharing networks by dynamically electing nodes to become super-peers, caching and serving common queries or content.

2.3.1.4 Third generation: efficient routing substrates

Although the range of applications for P2P techniques remained limited, by the end of 2001 a common requirement had emerged. In order for each peer to make a useful contribution to the global service, a reliable way of partitioning workload and addressing the responsible nodes was needed. Further, the emphasis on scalability, and the corresponding observation that in global-scale system peers will be joining, failing, and leaving continually, required these functions to be performed with the knowledge of only a fraction of the global state on each peer, maintained with only a low communication overhead in the underlying network. These observations inspired a generation of P2P routing substrates that provided a distributed message passing, object or key location service. The most popular approaches adopt a virtual address space, in which nodes are assigned a unique pseudo-random identifier that determines their position in the space (see figure 2.8). Messages are then routed toward keys in the same address space, and are delivered eventually to the node numerically closest. According to the way in which applications use this service, a message destined for a given key represents a request to provide a given service with respect to that key. As requests' keys must be mapped on to the key space pseudo-randomly, these platforms offer effective partitioning of the work between peers. Different variants of this basic approach differ as to the structure of information on nodes and the way messages (or sometimes requests for routing information) are passed between peers.

2.3.2 Unstructured and Structured P2P Networks

Unstructured P2P Networks (UPNs) appeared earlier to bring together edge resources. Filesharing applications are predominant in this context, where Gnutella [55] embodies UPNs. They are called unstructured because links between nodes are established arbitrarily. Nevertheless, UPNs suffer of two main drawbacks: the lack of query correctness and the overhead on communication. The former can be explained as follows: suppose there is a file in the UPN and a user wants to retrieve it. If the file is too far from the user, it might not to be found by the user query. The latter occurs because communication in UPN is performed mainly by flooding and causes a high amount of signalling traffic in the network. Hence, such networks typically have very poor search efficiency.

Structured P2P Networks (SPNs), such as Pastry [82], appeared to overcome the problems appeared in UPNs. SPNs are able to guarantee query correctness, a key property for modern applications, as well as routing and time efficiency. SPNs are also broadly called distributed hash tables (DHTs) because most of them associate the data owner node by means of a consistent hashing, in an analogous way to that

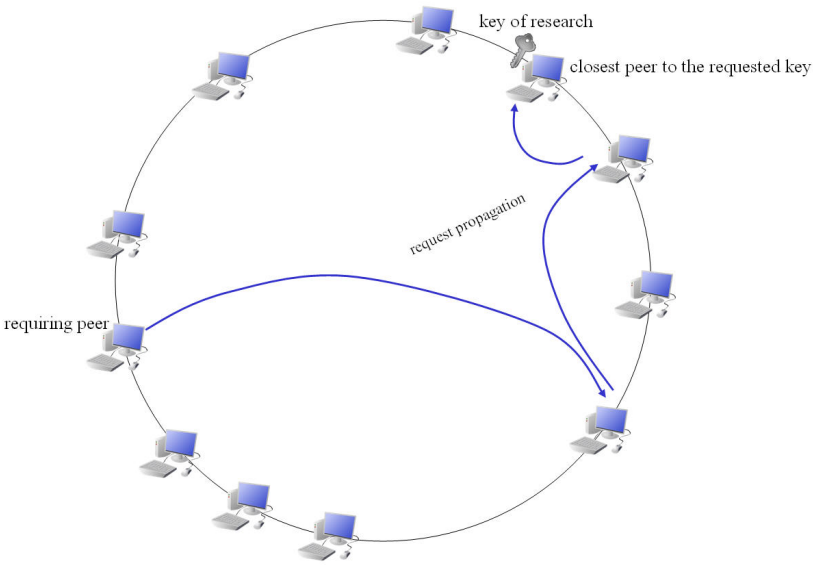


Figure 2.8. Structured architecture

of traditional hash table's assignment of keys to a bucket. SPNs provide exact-match queries with correctness and applications can use the API put(key, value) / $value \longleftarrow get(key)$ to access to the content.

2.3.3 Pastry

Pastry[9] is an overlay and routing network for the implementation of a DHT. The key-value pairs are stored in a redundant P2P network of connected Internet hosts. A Pastry network is characterized by its redundant and decentralized nature; in this way there is no single point of failure and any single node can leave the network at any time without warning and with little or no chance of data loss. The protocol is also capable of using a routing metric supplied by an outside program, such as ping or traceroute, to determine the best routes to store in its routing table.

Each node in the Pastry overlay is assigned a 128-bit node identifier (*nodeId*). The *nodeId* is used to determine the node's position in a hypothetical circular *nodeId* space, which ranges from 0 to 2^{128} - 1. The nodeId is assigned randomly when a node joins the system. In general it is assumed that nodeIds are a sequence of digits with base 16 and are generated such that the resulting set of nodeIds is uniformly distributed in the 128-bit nodeId space (see figure 2.9).

[9]http://research.microsoft.com/en-us/um/people/antr/pastry/

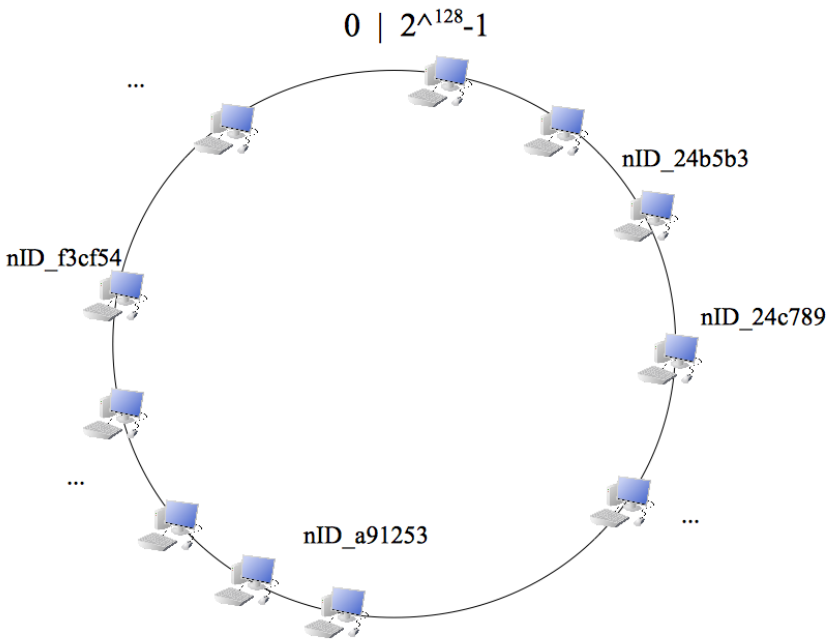


Figure 2.9. PASTRY nodeId distribution

Often, it is possible to generate nodeIds, e.g., by computing a hash function of the node's public key or its IP address. As a result of this assignment of nodeIds, with high probability, nodes with adjacent nodeIds are diverse in geography, ownership, jurisdiction, network attachment, etc.

The major function of Pastry nodes is to route messages (for instance to *store* an entry within the DHT) from one node to others. A node efficiently routes a message identified by a key to the node with a nodeId that is numerically closest to the key, among all currently live Pastry nodes. The expected number of routing steps is O(log N), where N is the number of Pastry nodes in the network. In each routing step, a node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit (or 4 bits) longer than the prefix that the key shares with the present node's id. If no such node is known, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's id. To support this routing procedure, each node maintains some routing state. Pastry takes into account network locality; it seeks to minimize the distance messages travel, according to a scalar proximity metric like the number of IP routing hops. Each Pastry node keeps track of its immediate *neighbours* in the nodeId space, and notifies applications of new node arrivals, node failures and recoveries. Each Pastry

node knows a certain set of other live nodes in the P2P network, through a *routing table* coupling nodeIds to the associated node's IP addresses of the known nodes.

To **join** the network the protocol requires a single node to bootstrap by supplying the IP address of a peer already in the network, than inform other nodes of its presence. To do so the new node could have an own IP list of potential live nodes. Otherwise, the information about neighbours can be obtained through outside channels such as well known IP addresses listed publicly on the Web. For instance, let's assume that a new node with nodeId X asks the already existing node with nodeID A to join the network. Node X then asks A to route a 'join' message with the key equal to X. Like any message, Pastry routes the join message to the existing node Z whose id is numerically closest to X. In response to receiving the 'join' request, nodes A, Z and all nodes encountered on the path from A to Z send their routing tables to X. The new node X then initializes its own routing table considering closest existing nodeId. Finally, it informs any nodes that need to be aware of its arrival. Pastry uses an optimistic approach to controlling concurrent node arrivals and departures. Since the arrival/departure of a node affects only a small number of existing nodes in the system, contention is rare and an optimistic approach is appropriate. Briefly, whenever a node A provides state information to a node B, it attaches a timestamp to the message. B adjusts its own state based on this information and eventually sends an update message to A (e.g., notifying A of its arrival). B attaches the original timestamp, which allows A to check if its state has since changed. In the event that its state has changed, it responds with its updated state and B restarts its operation.

The notion of network proximity is based on a scalar proximity metric, such as the number of IP routing hops or geographic distance. A node with a lower distance value is assumed to be more desirable. A Pastry node, to determine the 'distance' of a node with a given IP address to itself, is expected to implement this function depending on its choice of a proximity metric, using network services like traceroute or Internet subnet maps, and appropriate caching and approximation techniques to minimize overhead.

The efficient Pastry routing scheme allows to develop an Internet-based, P2P global storage utility called PAST [32] which aims to provide strong persistence, high availability, scalability and security. A storage utility like PAST is attractive for several reasons: i) it exploits the multitude and diversity (in geography, ownership, administration, jurisdiction, etc.) of nodes in the Internet to achieve strong persistence and high availability; this obviates the need for physical transport of storage media to protect backup and archival data; ii) it renders unnecessary the explicit mirroring of shared data for high availability and throughput; iii) it also facilitates the sharing of storage and bandwidth, thus permitting a group of nodes to jointly store or publish content that exceeds the capacity of any individual node.

A storage management scheme in PAST ensures that the global storage utilization in the system can approach 100%, despite the lack of centralized control and widely differing file sizes and storage node capacities. In a decentralized storage system where nodes are not trusted, an additional mechanism is required that ensures a balance of storage supply and demand.

File persistence in PAST is assured because the file is stored on several storage nodes. When a file is inserted in PAST, Pastry routes the file to the nodes k whose node identifiers are numerically closest to the 128 most significant bits of the file identifier (*key*). Each of these k nodes then stores a copy of the file. The replication factor k depends on the availability and persistence requirements of the file and may vary between files. A lookup request for a file is routed towards the live node with a nodeId that is numerically closest to the requested *key*.

This procedure ensures that (1) a file remains available as long as one of the k nodes that store the file is alive and reachable via the Internet; (2) with high probability, the set of nodes that store the file is diverse in geographic location, administration, ownership, network connectivity, rule of law, etc.; and, (3) the number of files assigned to each node is roughly balanced.

Chapter 3

Related Work

This chapter introduces the scientific and technological background of our research, which spans the areas of *Semantic Web* and *Distributed Index*. The chapter reports a discussion on related work. It concludes with the analyses of the requirements and the fundamental choices made by the author to guide the implementation.

3.1 Semantic Desktop

In [88], the idea of Semantic Desktop is described as follows: *A Semantic Desktop is a device in which an individual stores all her digital information like documents, multimedia and messages. These are interpreted as Semantic Web resources, each is identified by a Uniform Resource Identifier (URI) and all data is accessible and queryable as RDF graph. Resources from the web can be stored and authored content can be shared with others. Ontologies allow the user to express personal mental models and form the semantic glue interconnecting information and systems. Applications respect this and store, read and communicate via ontologies and Semantic Web protocols. The Semantic Desktop is an enlarged supplement to the user's memory.*

Many research projects are attempting to merge the focal parts of Semantic Web into desktop computing, P2P and Social Networking. The Gnowsis project [87][89] deals with the details of integrating desktop data sources into a unified RDF graph and identifies resources with URIs. The main idea was to enhance existing desktop applications and the desktop operating system with Semantic Web features. Whenever a user writes a document, reads e-mails, or browses the web, a terminology addressing the same people, projects, places, and organizations is involved. The terminology grows out of the interests and the tasks of the user. Gnowsis was aimed to be combined with web 2.0 philosophy and semantic web technologies as a useful basis for future semantic desktops but not as an integrated

web system for widespread use among internet communities.

Other projects focus on the issue of data integration, aggregating data obtained from the web. In the web services world, the SECO project [44] aims at integrating web sources via an infrastructure that lets agents uniformly access data that is potentially scattered across the web. Using a crawler, it collects the RDF data available in files. RDF repositories are used as sources for data. Integration tasks over the various data sources, such as object consolidation and schema mapping, are carried out using a reasoning engine and are encapsulated in mediators to which software agents can pose queries using a remote query interface. SECO includes a user interface component that emits HTML, which allows human users to browse the integrated data set. To create the user interface, a portion of the whole represented data set is considered and used to generate the final page. The structure of the pages of the site is created using a query, the results of the query are transformed to HTML, giving three fundamental operations: a list view, keyword search functionality, and a page containing all available statements of an instance.

3.2 Distributed Systems

Nowadays there are more tools than ever to help harness unused computing power in hundreds of personal computers. Traditionally, three categories of "distributed" computing existed: *Cluster computing*, similar machines (generally servers of similar power and configuration) are joined to form a virtual machine. Linux clusters are good examples; *Peer to peer*, many desktop computers are linked to aggregate processing power. The distinguishing characteristic is the machine itself, which almost exclusively is a low-power client PC. Often, the link is via Internet; *Distributed computing*, increasingly known as grid computing, this approach connects a wide variety of computer types and computing resources, such as storage area networks, to create vast "virtual" reservoirs of computers to serve geographically separated users.

The traditional client-server internet model gives some ground to P2P networking, where all network participants are approximately equal. The primary advantage of P2P networks is that large numbers of people share the burden of providing computing resources (processor time and disk space), administration effort, creativity and legal liability. It is relatively easy to create a community of users in such an environment and it is harder for opponents of a P2P service to bring it down.

Distributed Hash Tables (DHT) are considered a key technology in P2P applications as a consequence of their robustness and scalability. Several projects [25][81], as well as popular file sharing applications make use of DHTs in order to distribute the data over a large number of peers, that contribute storage to a community of users. In the last years, the research and development in the P2P field has been considerable. Napster, Gnutella2, Edonkey2K, Bit Torrent, Kademlia [60] are only

a few examples of consolidated protocols/architectures. The use of a such infrastructure in general is justified by it desirable features for P2P systems, such as resistance to the censorship, decentralization, scalability, security, etc. To distribute data among a thousand or a million of peers not only involves a huge amount of information, but also confidence in a robust system that is free of restriction from a central authority. This enables virtual communities to self-organize and introduce incentives as a resource for sharing and cooperation. One can therefore argue that what is missing from today's P2P systems should be seen both as a goal and a means for self-organized virtual communities to be built and fostered.

Between 2001 and 2002, almost simultaneously several architectures for DHT were born, such as CAN [79] , Chord [96] , Pastry [81], Kademlia [60], and others.

3.3 Distributed Index

Whenever we think of distributed systems, we have to have think also about the way to catalogue the information. During the last few years there has been a growth in the number of distributed indexes for P2P networks, and a closer connection with the ontology field.

In SA Net [86], an agent-based system achieves its semantic richness through the use of explicit ontologies to represent resources. SA Net enhances the DHT based resource distribution scheme by using a unique identifier assigned to each ontology as a key to locate the overlay node responsible for maintaining the resource index associated with the underlying ontology. In other words, the ontology-based hashing scheme utilizes ontologies, instead of resource names, as the hash input to generate the key necessary to distribute the resource among overlay nodes.

The SCORE project [93] provides classification and terminological basis for contextual reasoning on metadata. Metadata are divided into two types: syntactic metadata and semantic metadata. Syntactic metadata describe non-contextual information about content, e.g. language, length, date, audio bit-rate, format, etc. Such metadata offer no insight about the content. Semantic metadata describe domain-specific information about content.

The PAGE [101] (Put And Get Everywhere) project consists of a peer to peer infrastructure for distributed RDF storing and retrieval. It starts from YARS [45] , a solution that defines an optimized index structure for fast retrieval of RDF statements. PAGE implements YARS index structure that indexes resources using RDF encoding called quad (spoc), where s is the subject, p the predicate, o the object and c is the context. The index is achieved creating keys becoming from the combination of quad elements.

RDFGrowth algorithm [99] introduces a scenario of a peer to peer network where

each peer has a local RDF database and is interested in growing its internal knowledge by discovering and importing it from other peers in a group. It is important to consider this kind of approach in order to define a mechanism of queries based on SPARQL formalism. That type of queries requires an access to an RDF knowledge base. The problem is how to distribute a centralized knowledge base on different nodes in order to satisfy a query by accessing only one node.

3.4 Semantic Indexing

Atlas [53] is a P2P system for storing, updating and querying RDF(S) data and RDFS ontologies. It is part of the OntoGrid[1] project, where it has been used as a distributed repository for RDF(S) metadata describing Grid services and resources. Atlas focuses on E-science data repositories semantically annotated with RDF. The challenge is the development of efficient indexing techniques and relational-style statistics-based query optimization. Existing RDF stores have excellent performance, but lack when used in wide area networks applications such as content-sharing, Web(Grid service registries, distributed digital libraries, and social networks. Strategies for RDF query processing are based on two kinds of queries: i) *one-time queries*, standard queries such as those asked in a typical RDF store and the answer retrieved only once after the query is executed; ii) *continuous queries*, long standing queries that return a response every time a relevant update is executed. In the Atlas system, the *metadata provider* supplies resource descriptions by RDF(S) data. The *metadata consumer* wants to discover resources by submitting SPARQL queries to the system. To store RDF(S) data, it is necessary to submit it to the system in the form of an RDF(S) document. The document is decomposed into a collection of RDF triples and *triple-indexing* is performed. Each triple, of the form (subject, predicate, object) is stored in the system three times, once with the subject as the key, once for the predicate and one for the object. Each node of the P2P network stores the triples it receives in its local database (based on SQLite[2]). For answering a query, the node that poses the query is responsible for parsing it and producing an internal SPARQL representation. The query is evaluated by a chain of nodes. Intermediate results flow through the nodes of the chain and finally the last node in the chain delivers the result back to the node that submitted the query.

[1] http://www.ontogrid.net/
[2] http://www.sqlite.org

3.5 Discussion of Related Work

Long time ago, Vannevar Bush, the director of Office of Scientific Research and Development of United states, wrote an article in The Atlantic Monthly Journal titled "*As we may think*" [21] , describing the idea of an hypertextual machine called Memex, a kind of electronic desktop equipped with a microfilm storage memory, allowing one to save pages of books and documents, play them and associate with each other to make knowledge more accessible. The essay predicted many kinds of technologies invented after its publication. Progress in Semantic Web, P2P and natural language processing has resulted in new forms of collaboration and social semantic information spaces. We are interested in investigating what the scientific community has proposed so far and in innovating the processes of semantic indexing in distributed systems with respect to P2P networks where the index is scattered among a DHT. We investigate semantic indexing according to domain specific ontologies and aim at providing the user with specific and easy-to-use tools for browsing ontologies for choosing significant elements for index keys. In order to publish and retrieve the knowledge, it is necessary to use complex reasoning and inference on the semantics of the published information. For creating a network of users in the sense of a social network, it is necessary to create a platform similar to a semantic desktop that is equipped with a set of tools at a user's disposal. Even Tim Berners Lee did not really envision the World Wide Web as a hypertext delivery tool, but as a tool to make people collaborate [31]. Our system deals with technologies that combine P2P networks, semantic indexing and domain specific retrieval systems. In recent ONTOSE conferences (International Workshop on Ontology, Conceptualization and Epistemology for Software and System Engineering) a lot of attention has been payed on this topic. Progress in Semantic Web, peer to peer networks and natural language processing, leads to new forms of collaboration and new social semantic information spaces, including the following:

AgentSeeker [68] is a multi-agent search engine for managing company knowledge bases. An ontology agent is devoted to managing the enterprise domain in a semantic way. The goal is to make document retrieval a more intelligent process, finding texts which are semantically bound to the user's query. The core behaviour of *AgentSeeker* is to parse text files and to keep a database for storing extracted information. Every record includes the URIs of the ontologies supported and a measure of the affinity, in terms of percentages of words of the document that are also contained in the ontology. *AgentSeeker* has only the textual content available because the authors' opinion is that the tagging of the resources becomes complicated due to the impossibility of modifying a file or difficulty in manually cataloguing thousands of documents. In case of specific domains, which *AgentSeeker* considers, it is not prohibitive to think of manually cataloguing personal resources if specific tools are provided. It is not necessary to modify the file content because a semantic

description can be associated with the file. In our approach we provide such tool that associates a semantic description to any kind of document. As stated in [80] , ontologies allow adding semantics to data so that different software components can share information in a homogeneous way. Furthermore, logic can be used in conjunction with such formal representations for reasoning about information and facts represented as ontologies. We also take this in consideration in our description of the expansion of the system.

In the field of distributed knowledge management, *SA Net* [86], an agent-based system, achieves its semantic richness through the use of explicit ontologies to represent resources. SA Net further enhances the DHT based resource distribution scheme by using a unique identifier assigned to each ontology as a key to locate the overlay node responsible for maintaining the resource index associated with the underlying ontology. In other words, the ontology-based hashing scheme utilizes ontologies instead of resource names as the hash input to generate the key necessary to distribute the resource among overlay nodes. In our approach we give the same responsibility to all nodes of the network. We have a slightly different meaning of semantic indexing in that we do not directly attach resources to ontologies but create keys whose content refer to ontologies. In this way there is a symmetric approach for keys creation in both publishing and retrieving resources in the DHT.

The *SCORE* project [93] provides classification and terminological basis for contextual reasoning on metadata. Metadata are divided into two types syntactic metadata and semantic metadata. Syntactic metadata describe non-contextual information about content, e.g. language, length, date, audio bit-rate, format, etc. Such metadata offer no insight about the content. Semantic metadata describe domain-specific information about content. We are of the opinion that meta-data are not sufficient for describing resources and allowing some reasoning upon them. We do not intend to extract or use metadata from different structured information sources.

RDFGrowth algorithm [99] introduces a scenario of a peer to peer network where each peer has a local RDF database and is interested in growing its internal knowledge by discovering and importing it from other peers in a group. It is important to consider this kind of approach in order to define a mechanism of queries based on SPARQL formalism. That type of queries requires an access to an RDF knowledge base. The problem is how to distribute a centralized knowledge base on different nodes in order to satisfy a query by accessing only one node. In our scenario the DHT stores the distributed knowledge base. The process starts by browsing several ontologies that a user can index or search for resources. The system is building indexing keys in the background. The types of allowed requests determine the types of indexing keys and routing algorithms. In a centralized case, a compound query is an investigation on a knowledge base (looking for the triples which suit the query in RDF bases). In our case we have to face issues of a distributed knowledge base. A direct query of the whole knowledge base is impossible.

Our work aims at demonstrating the benefit of a semantic indexing engine exploited by a set of tools available for a community of users located in different places. To create a community of users means to tackle the topic of distributed systems. The first requirement is to have systems independent from any central point of aggregation. Among distributed systems, P2P architecture brings most advantages, among them decentralization, scalability, fault tolerance. It is mandatory to have an efficient distributed data structure to efficiently store and retrieve elements from a huge amount of information; the evident efficiency of DHTs relies on the number of messages exchanged to route a query to its destination. The order of magnitude of this number is O(log(N)), where N is the total number of nodes. In the present work, the low-level layer of P2P applications is built on FreePastry (see 2.3.3 for details), the open-source implementation of Pastry, whose significance is guaranteed by the support provided by the community of users regularly improving and amending it; its features allow for adapting the network to the specific needs of users. We do not need to deduce new metadata from different structured information, but simply to create an index whose content refers to ontologies. However some reasoning elements have to be taken into account.

From a user's point of view, it is necessary to access a set of tools that provide an intuitive interaction. The development of Gnowsis started with enhancing desktop applications with the features of Semantic Web, but only standalone applications were considered. Considering the web 2.0 approach and using semantic web technologies, we have created an integrated web system, similar to a common desktop, for a large use among internet communities. As experienced in the SECO project, many RDF-compliant heterogeneous data sources are queried and depicted to the user via ad-hoc web user interfaces generation. Our purpose is not to aggregate data obtained generally from the web but from a community allowing to limit the context of data and to ensure better adequacy for research goals and results.

Using ontologies in distributed systems like P2P networks, is regarded favourably by the scientific community. In 2002, the Edutella project [65] handled by Sun Microsystems, made a first approach for the association of semantics to educational content through an open source infrastructure based on RDF metadata for the interoperation between different schemes (IEEE/LOM, IMS Learning Design[3] , ADL SCORM), and performed a mapping among them. The SWAP project [34] , managed by the University of Karlsruhe, pays special attention to the topics related to Semantic Web. Its aim is to allow computers to actually comprehend the meaning of its processed data. Using the model of ontologies, the project allows to develop a technology in the area of knowledge management and P2P. Complex structures can be easily encoded in a set of RDF triples. In [101] it is argued that RDF should become the bases of the Semantic Web. Nevertheless, RDF is not enough; it does not

[3]http://www.imsglobal.org/learningdesign/index.html

supply sufficient expressive power to represent the whole knowledge schema. DHT-based overlay systems offer an interesting alternative to existing information system architectures. We propose to express the semantic classification through concepts articulated by ontologies that describe the specific domain and to formulate such expression by the OWL formalism.

3.6 Requirements and choices

In P2P networks, resources are generally stored and indexed in a distributed hash table (DHT [58][79][82]), and in personal memory, resources should be stored using the same kind of index. A P2P architecture avoids both physical and semantic bottlenecks that limit information and knowledge exchange [95]. The success of such networks using semantics was considered challenging in [30]. Using the DHT, overlay nodes can be identified deterministically and in a finite number of steps (normally the complexity is log(n) where n is the number of nodes of the overlay), thereby reducing considerably the search overhead due to communication and routing. One critical limitation of DHT-based structure, with respect to resource discovery, is its lack of support for location-based queries that go beyond perfect matching. This has a strong influence on the building of index keys. Resources are semantically indexed using ontologies. We follow the W3C recommendations and use OWL as the ontology representation language. A resource description is a set of assertions that may involve several ontologies. In particular, a resource is considered an instance of one or more concepts that can be found in some ontologies. For example, in e-learning context, a resource may be considered a learning object. An entry key of the index is based on the assertions that denote a resource. Ontologies used for indexing must be stored in the shared memory [42][40] and later retrieved.

Chapter 4

Research

This chapter presents theoretical results of the study. It gives definitions of relevant concepts and explains relations between them. Aspects of research concerning *Semantic Indexing*, *Resource Description*, *Creation of keys*, *Use of Ontologies* and *Community* are explored and shortcomings in the state of the art approaches in the above-mentioned fields are addressed. An attempt is made at contributing to the open issues.

4.1 Semantic Indexing

4.1.1 Introduction

The creation and sharing of documents is a daily concern for several categories of community users. We consider people belonging to loose communities and relying on a P2P network. The nature of communities is not relevant because the solution we propose is generic. Members of a community usually manage their private resources and agree to share a part of them with the other members. For that they use private and shared memory, respectively. Users need to describe the resources. It seems more convenient to use the same system of description for the resources stored in the two types of memories. The advantage is that users can retrieve a resource from the network and/or the private storage thanks to a common search system. Considering the P2P nature of communities it is necessary to create an index of managed resources. Indexing is the process of creating or updating an index. Starting with a list of resources allows one to create a correspondence between identifiers and resources. There are many examples of indexes. E.g. the index of a book is the association between book parts (identifiers) and page numbers (resources). We generally consider two kinds of models for indexing resources: *boolean* [85] and *vectorial* [84]. As a *boolean* model, the index of documents is an inverse file

which associates to each keyword of the indexing system the set of documents that contain the keyword. In a *vectorial* model, a document is represented by a vector whose dimensions are associated to the keywords occurring in the document and the coordinates correspond to the weights attached to the keywords in the document thanks to a specific calculus. A request is also a vector of the same nature. The answer is composed of a list of documents presenting similarities with the request thanks to a specific measure [14] [92] based on coordinates of the vector.

In centralized indexing [76] [102] both models are available. However, in the case of P2P networks, the index must be distributed among the peers and the number of queries sent to the system when searching for resources should be minimized, because they are time consuming. Due to these constraints, the model of a distributed index is necessarily *boolean*.

Documents may have different types (text, audio, video, archive, etc.) and different storing formats. To share and retrieve resources, it is necessary to give them a proper description. The traditional file sharing diffusion is based on the title of the resource. We cannot consider the title as the right way to identify a resource because in our case, resources are created locally and their meaning is not universally known by their titles. So it is necessary to find another way for describing resources.

For that, we can consider some sort of tagging which consists of associating keywords to resources. In many cases, keywords make descriptions ambiguous, as in the following example:

the resource is entitled *An interesting overview of Java.* The keyword *Java* is used for description.

What does it mean? Does *Java* refer to the island of Indonesia or does it refer to the programming language developed at Sun Microsystems? Without taking into account the contextual elements, the question has multiple answers.

This solution is not satisfactory because we cannot permit that the system based on this description type accepts too many ambiguous cases.

To deal with this difficulty, we propose a very fine mechanism of description of the resources. We consider semantic descriptions built from the elements extracted from ontologies. The mechanism of description and the semantic indexing technique are parts of our system (hereafter with the expression *"the system"* or *"our system"* we refer to the theoretical model described in this work and implemented in our prototype).

The Semantic Web proposes standards for allowing machines to understand the meaning (or "semantics") of information. It defines a set of well supported languages such as RDF, RDFS, OWL, SPARQL, and related technologies. In our solution, semantic information strictly related to a resource is written in a language based on RDF and is included in its description. We have chosen to use domain ontologies written in RDFS or OWL. We do not consider the design of ontologies but we assume to find well structured and ready to use ontologies. Manual semantic indexing is

possible only with named concepts with clear descriptions.

With respect to the previous example, the choice of the right ontology allows to know if *Java* can be attached to a geographical concept or to a computer science one.

Definition 4.1. *(Semantic Description)*
We call Semantic Description the representation of a resource that uses ontological elements.

In our case, a Semantic Description is manually constructed but a part of the Description could be done automatically with applicable software. In figure 4.1 we can see that the system uses a formal representation of the resource, based on user input and available ontologies. The description is prerequisite for indexing.

Definition 4.2. *(Semantic Indexing)*
We call Semantic Indexing the association of a Semantic Description to a resource.

The figure 4.1 shows the meaning of an association. The same description may be used to identify several resources.

Semantic indexing is defined with respect to the possible queries that the system can answer. A query provided by users is a logical expression combining semantic descriptions and boolean operators (described in section 4.1.4.9). The system answers with the list of documents that satisfy the logical expression. The semantic descriptions proposed by the user are supposed to be contained in the index.

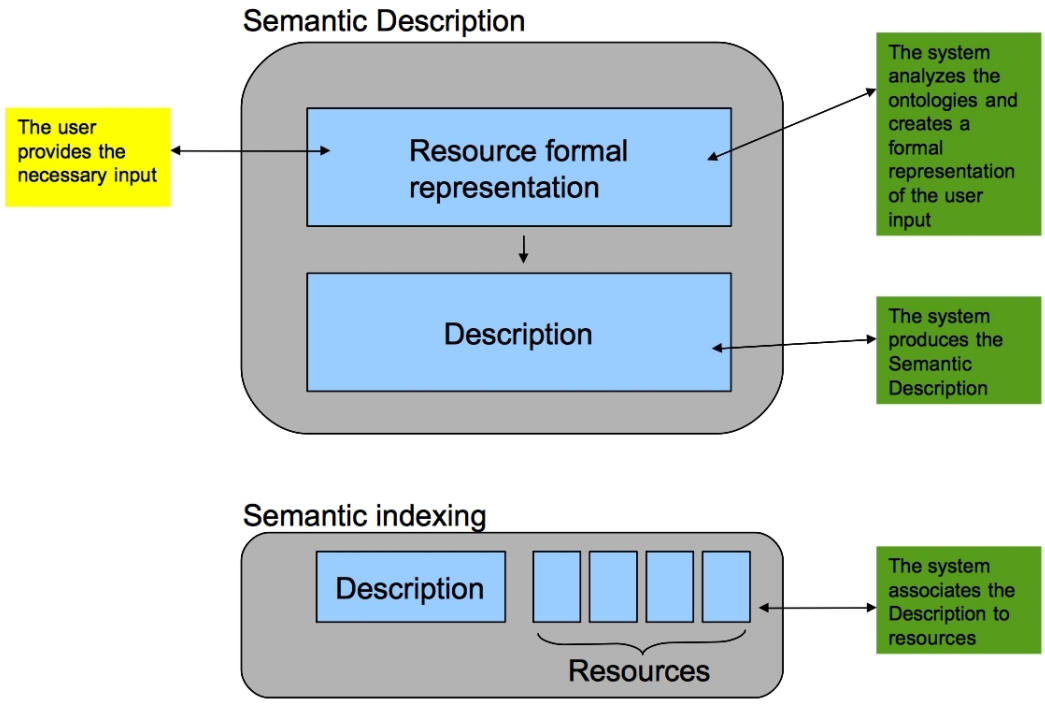


Figure 4.1. Semantic Description and Indexing

4.1.2 Approach

4.1.2.1 Non-semantic approach

Folksonomies are defined as Internet based collaborative systems for non hierarchical and spontaneous categorization and organization of web resources through shared tags[98] [23]. Users can freely add descriptive keywords to resources. A system can understand the textual content trying to navigate through the system of labels assigned to different resources by the users that participate in a folksonomy.

In folksonomies, the query system consists in simple navigation between registered labels and accessing correspondingly tagged pages.

In most cases the user produces a query containing a sequence of keywords and expressions combined with boolean operators (AND, OR, NOT). The system answers by searching within an index for resources satisfying the logical expression. However, this approach does not take into account synonyms or contextual elements.

4.1.2.2 Semantic approach with free text input

In ontology engineering, it is necessary to answer questions related to modeling practice. Among them we find: *Who does what, when and where?*, *What are the parts of what?*, *What is an object made of?* [37].

In query engineering, studies have to be managed in order to define the types of queries a user can produce. Common queries may have a general purpose like: *What are the documents about the subject Medieval Italy?*, or *What are the documents published by a specific author in 2008?* Some queries seem general but infact refer to a specific domain, e.g. *Theory of Languages*, such as: *What are the documents written by Chomsky?* (Chomsky refers to *Theory of Languages*).

Two queries may be dependent. For example, in the field of programming languages it is usual to refer to concepts concerning data structures like *stack, list or tree*. Queries like *What are the resources that concern stack?* and *What are the resources that concern data structures?* are not independent when a relation between the concepts of stack and data structure exists in the ontology chosen to describe the domain to programming languages. In some cases, a resource can be described in different ways, so the same resource can be retrieved by several different queries.

Even if we consider semantic descriptions of resources, there are some different ways a user can address a query to a system. Moreover user interface of the system must be designed in line with the query model accepted by the system. Let's assume a system that accepts domain ontologies. We may consider different ways the user can input a query in a system:

- the user produces a query written in natural language: e.g. *What are the resources that concern stack?*
- the user produces a query containing a sequence of keywords

These two cases require textual analysis of the query in order to identify the concepts and relations of an ontology that can be associated to the content of the query [63].

The necessary matching between such query content and ontology content is based on text analysis techniques such as those supplied by Apache Lucene [46]. Sometimes a query analyser is able to detect incomplete formulation and may begin a dialogue [78] with the user for adding missing elements.

This approach has some limitations and is not suitable for our aims. User input could produce noise due to typing errors. The free use of keywords could bring ambiguity or imply the need for very sophisticated text analysis.

We propose a third approach based on *description patterns* 4.5. The user does not formulate the query herself but is guided by a system for building it. The system creates in the background the real query in a specific internal language and sends it to the network.

4.1.2.3 Our Semantic Approach

In our system we do not introduce any specific query language. Based on the concept of *pattern*, the system provides the user with a method for creating resource descriptions in an interactive way. The description is based on potential queries that the system can answer. Hereafter we denote by the word *description* the semantic description that a user gives to resources. We say "*the user thinks of the query*", meaning that when the user creates a description, she thinks about possible queries that the system can answer. Moreover, when we say "*the user builds a query*" we mean that the user creates a query to get an answer from the system. In fact, the description is the information that is useful in the system for indexing resources, for storing and for retrieval. The user thinks of the query in "natural language". The system, considering the input from the user, applies a mechanism of *reformulation* of the query based on the available ontologies.

The user must first identify the type or category of resources she is looking for. This decision is made choosing the proper ontology related to the domain of the resource. It is the ontology that contains at least one concept which denotes the resources the user is interested in. The approach considers resources as instances of well identified concepts. The system proposes a list of concepts which constitute the *entry points* of the ontology (see section 4.4). Once the initial concept is chosen, a list of properties related to the type of resources is proposed to the user. The process goes on, following step by step a path in an ontology and stops when the end of the path is reached. From the steps followed, the system creates a small knowledge base. See section 4.3 for details on cycles problems.

For instance, the user thinks of the following query: *What are the documents written by Chomsky?*. This step is identified as "Query in NL" (NL means "Natural Language") in figure 4.2. It is not actually part of the creation of the "Semantic Desription", but it is just a mental process the user does before providing input to the system. Figure 4.2 shows that, starting from the user formulation, the system creates a formal representation of the user input, paraphrasing the user query. From the formal representation the system creates the description.

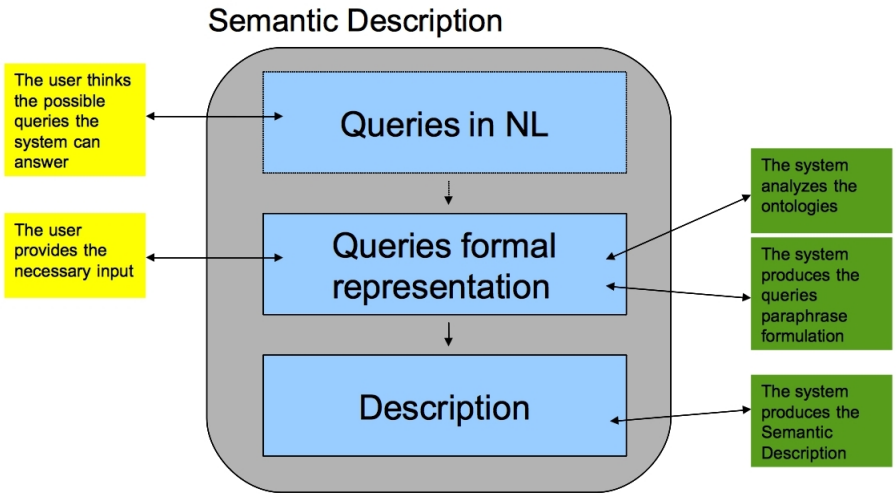


Figure 4.2. Query paraphrase formulation

The system guides the user in navigating the chosen ontology, recognizes the minimal path of the ontology that represents the meaning of the user query and builds the real query. In the example, the real query can be translated in natural language by *What are the documents having an author whose name is Chomsky?* (that paraphrases the path in the ontology).

Figure 4.3 shows the path followed by the user and the complementary information found in an ontology (or in a knowledge base).

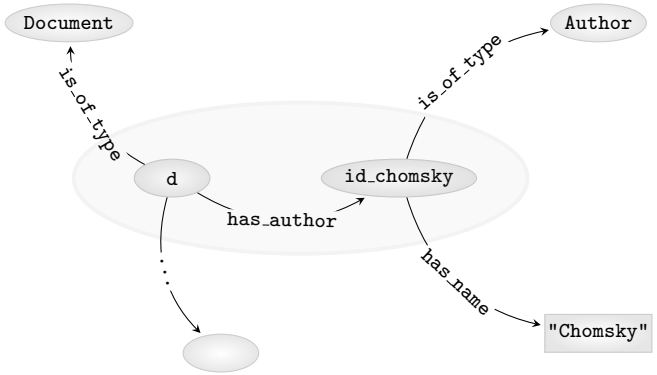


Figure 4.3. Query formulation

A path is a sequence of nodes representing individuals or concepts linked by arcs representing properties of the ontology.

4.1.3 Ontologies and Knowledge Bases

Our approach is based on the description elements that the user can select for describing a resource or building a query. They are contained in ontologies as concepts, relations and individuals.

We assume that expert community members are able to browse the Web in order to find the ontologies that can be used to describe the resources shared by the community. Eventually, they can build themselves some of the ontologies. They may also have the desire to build a population of an ontology grouping together with the most prominent individuals of the domain. In this case it is better to talk about knowledge bases available for describing a resource. For simplicity reasons, we only use the term "*ontology*" in the following sections to denote both an ontology and a knowledge base. The choice of more than one ontology does not introduce any ambiguity because each ontology is universally identified by its URI[1].

The ontologies used for resource descriptions are then published by the expert members in the P2P network as resources. Our system allows publication each time a new ontology becomes useful for the community, so it can be shared. However, most users are not aware of the existence of ontologies and are not involved in this process. The publishing of an ontology also requires a small description and an application domain for helping a user to choose the needed ontology.

It is not possible to cancel an ontology because the resources described with it could then not be retrieved.

We have created a system ontology (called *System Ontology*, denoted by *system*, detailed in section 4.4.2) for implementing some specific cases of resources description. It allows to identify an ontology that will serve for indexing resources. A special key, using the system ontology, is created for the publication and the discovery of the ontologies in the network. In particular the system ontology is also published in the network.

In some cases, a resource to be indexed must be considered as an instance of a specific concept (e.g. *Resource*, *Document*, *LearningObject*). Expert users have to recognize a particular set of concepts that can represent a type of resource when they want to publish a new ontology in the network. This set, called *Ontology Entry Points*, is then added to the description of the ontology. In our system ontology *entry points* are well emphasized in order to give the user the awareness of those elements denoting the resources the ontology can help to retrieve. In describing a resource, they constitute the first element the user has to select.

[1] http://www.w3.org/Addressing/

During the process of indexing, our system needs to look for the range of a property. Generally it is explicitly defined by the ontology designer and represents the most specialized concept, a type of the object (c) associated to a property (p) in triples like (s,p,c). To apply the *rdfs* semantics, if *(p, rdfs:range, C)* and *(C, rdfs:subClassOf, B)* then *(p, rdfs:range, B)*. So, a query asking for the range of a property will return a set containing the main specialized concept but also all its super-concepts. Due to this fact, we do not apply the *rdfs* semantics. In practice, as we will see in chapter 5, this problem is simple to solve. It is sufficient to use a Knowledge Base engine (Jena Model) that does not apply such kind of semantics.

The semantics of *rdfs:range* is depicted in figure 4.4.

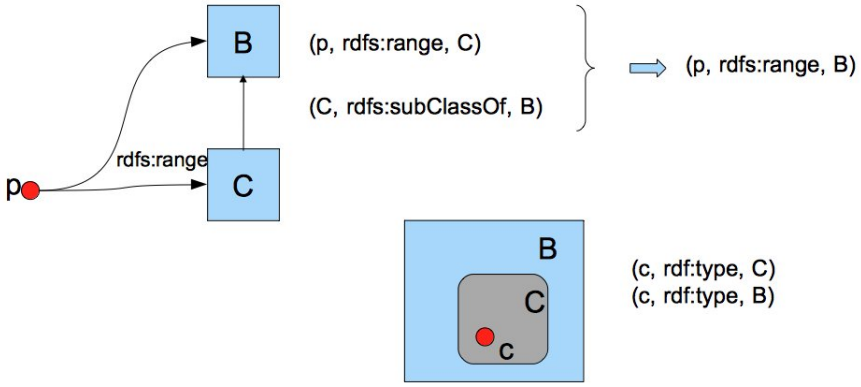


Figure 4.4. Semantics of rdfs:range

As shown in the following sections, the *System Ontology*, provided by the system, unifies the modelling of resource descriptions.

Hereafter we refer to an ontology by a namespace prefix that shortens its namespace name. In the context of our work, the namespace prefix identifies unambiguously the elements of an ontology.

4.1.3.1 The Set of Ontologies

For describing different examples presented in this work, we use the following set of ontologies: lom.owl (denoted by *lom*) [38], developed at "Université de Technologie de Compiègne" for representing the domain of learning objects; lt.owl (denoted by *lt*) describes the concepts of the Theory of Languages; system.owl (denoted by *system*) is an ontology we have developed for representing the resources of our system that allows indexing; tg-release3-1.owl (denoted by *tg*) [64] concerns social care services in the e-government domain; foaf.owl (denoted by *foaf*), the FOAF formal vocabulary description; GeoSkills.owl (denoted by *geo*), concerns the competencies, topics, and educational levels of the mathematics curriculum standards throughout Europe.

4.1.4 Types of Queries

4.1.4.1 Fundamentals

We consider two fundamental types of queries. The first one (called *Resource Query Type* in the rest of the document) aims at finding resources from elements of description that concern the resources themselves and not directly their content. For instances: *What are the resources written by Chomsky? I search for difficult exercises.* We consider that the resources concerned by the queries of this type can be represented by instances of a specific concept of a domain ontology: about *Theory of Languages*, denoted by *lt*, for the first example and about *Learning Object*, denoted by *lom*, for the second one. We call *Resource Type* the initial concept that the user must identify for denoting the resources she is looking for. The properties useful for interpreting user queries have to be chosen in the ontology of the resource type. This kind of queries relies on only one ontology.

The second type (called *Content Query Type* in the rest of the document) aims at finding resources from elements of description that concern their content. For instance: *What are the resources about Chomsky? What are the resources about grammar?* Both queries concern the topic of the content of the resource. The queries ask for a resource *about* something and not *written by* somebody. In most cases, it is not possible to find a unique ontology for describing the essence of a resource and its content simultaneously. We need a specific ontology, the *System Ontology* (see section 4.4.2 for details). Instances of the Document concept of the System Ontology represent the resources related to the queries of this type. The *system:Document* is the specific concept we use for the last two examples. Elements of a domain ontology (for the last two examples *Theory of Languages* and *Learning Object*) can represent the content of the resources associated with the queries of this type. These kind of queries rely on two ontologies.

4.1.4.2 Resource Query Type

Resource Query Type aims at finding resources giving elements of description concerning the resource itself and not its content. Let's consider the query: *What are the documents written by Chomsky?*

Our system helps to discover the concept that the resources are instances of. In this case the user chooses the concept of *Document* (*lt:Document*, among the ontology entry points proposed by the system) occurring in the ontology about *Theory of Languages*. The ontology contains a property whose domain is the concept *lt:Document* and that denotes the meaning of the query: *lt:has_author*, translating *written by*, leads to the *Author* concept.

Documents written by Chomsky.

paraphrases

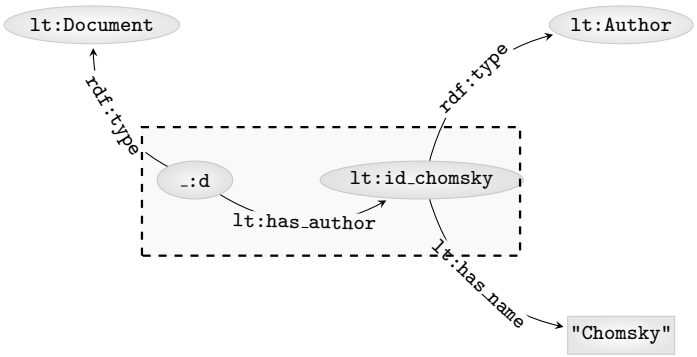


Figure 4.5. Representation of the query: *Documents written by Chomsky*

Documents has_author id_Chomsky.

The process goes on following the next steps in the ontology. The ontology contains some individuals of the concept *lt:Author*. The system shows these individuals and the user can choose *lt:id_chomsky*, verifying with the displayed attributes that it is the right author (*"Chomsky"*). The process is finished and the end of the path is reached. In this case the choice of a property and an individual has determined all the elements contained in the user query.

Figure 4.5 shows the description of a resource satisfying the query. The rectangle delimits the path that the system uses to build the real query.

4.1.4.3 Content Query Type

Content Query Type aims at finding resources giving elements of description that concern their content.

For instance, the user imagines the query: *What are the documents about the author Chomsky?* She is looking for resources whose content gives information about the author Chomsky.

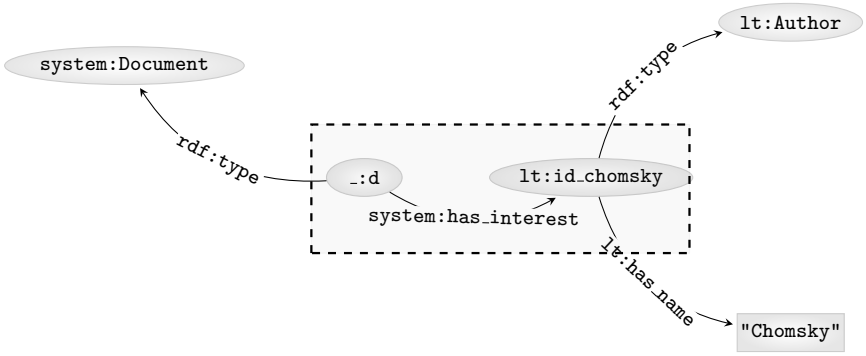


Figure 4.6. Representation of the query: *Documents about Chomsky*

These kind of queries require two ontologies. The system shows the entry points of the *System Ontology*. For the query of the example, the user chooses the concept *system:Document*. Instances of the concept *system:Document* represent the resources concerning the queries of this type. Among the properties of the System Ontology the user selects *system:has_interest* because its domain contains the concept *system:Document*. For representing the content of the resources concerned by the query, it is necessary to have a second ontology: the domain ontology *Theory of Languages*. The system shows the possible elements of a second ontology that can be contained in the range of the property *system:has_interest*. The user selects the individual of the concept *lt:Author* identified as *lt:id_chomsky*. This individual has for name *"Chomsky"*. The process is finished and the end of the path is reached. The elements contained in the user query have been determined by the choice of a property and an individual.

Figure 4.6 shows the ontological elements related to the query. The rectangle delimits the path that the System uses to build the real query.

Now, let's consider the query: *What are the documents about Grammar?* The user is looking for resources whose contents give information about the topic *Grammar*.

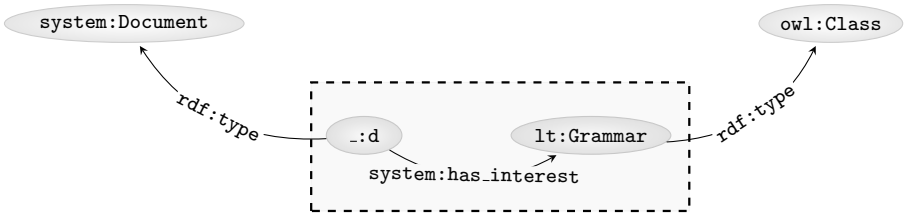


Figure 4.7. Representation of the query: *Documents about Grammar*

The first step of the process is the same as in the former example: the user selects an individual of the *system:Document* concept and the *system:has_interest* property. Then, the user chooses the second ontology about the *Theory of Languages*. In this ontology the user finds and select the concept *lt:Grammar*. The end of the path is reached and the process is finished. The elements contained in the user query have been determined by the choice of a property and a concept.

Figure 4.7 shows the ontological elements related to the query. For building a real query, the system uses the path delimited by the rectangle.

4.1.4.4 Open and Closed Queries

Our approach allows us to build *closed* queries i.e. all their elements are available in the ontology (or in the knowledge base).

However we have considered a particular case of an *open* query where the user can add the value of a property.

For instance, if the user intends to ask: *What are the documents written by XXX?*, where the *XXX* value is not in the knowledge base, she is allowed to add it. Figure 4.8 shows that the user has added "D.Ullman" as an author name.

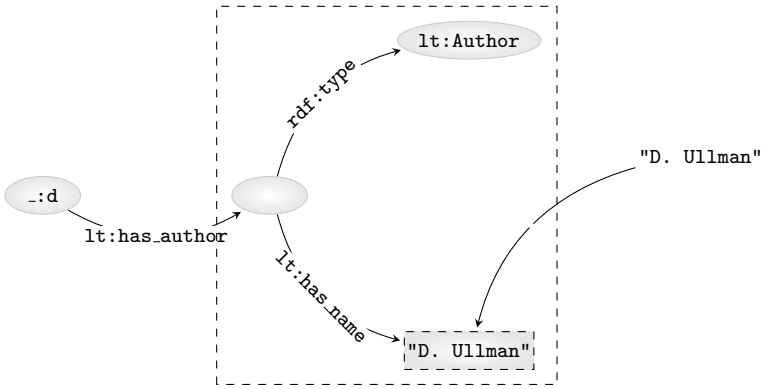


Figure 4.8. Knowledge base associated to an open query

With open queries, we give the user the possibility to input a keyword manually. It is important to observe that such keyword is not out of any context but it is provided as a value of a property of an ontology which is necessarily the last step of the query building process. Such a query may have success if and only if a description using the same keyword has been supplied by some resource provider.

4.1.4.5 Other Queries

Other queries are more complex because they concern a path involving more nodes.

Let's consider the following query: *What are the very difficult documents?* The query aims at finding resources with elements of description concerned with the essence of the resources themselves. This is the case of "Resource Query Type". The meaning of the user query refers to those resources that are documents; the documents are identified as those with an attribute of difficulty; the level of difficulty is "very difficult". The ontological elements required for interpreting the user query rely on the domain ontology *Learning Object Model, LOM* (denoted by *lom*). Among the entry points the system proposes, the user selects *Learning Object*. It is the concept whose individuals can represent resources searched by the user. For expressing the meaning of difficulty, the LOM ontology provides the concept *Difficulty*. It is not possible to find in this ontology a direct connection between an individual of *Learning Object* and an individual of *Difficulty*. In fact there is no property within the LOM ontology whose domain is the *Learning Object* concept and range is the *Difficulty* concept. A path with more steps is necessary.

After the user has chosen the entry point *Learning Object*, the system proposes the properties whose domain is the *Learning Object* concept. The user selects *has_lomEducational*. This property has for the range the concept *LomEducationalCategory*. In the next step, the user selects the property *has_difficulty* and an individual of the class *Difficulty* as the range of this property. This individual bears the label *"Very difficult"*. The process is finished and the end of the path is reached.

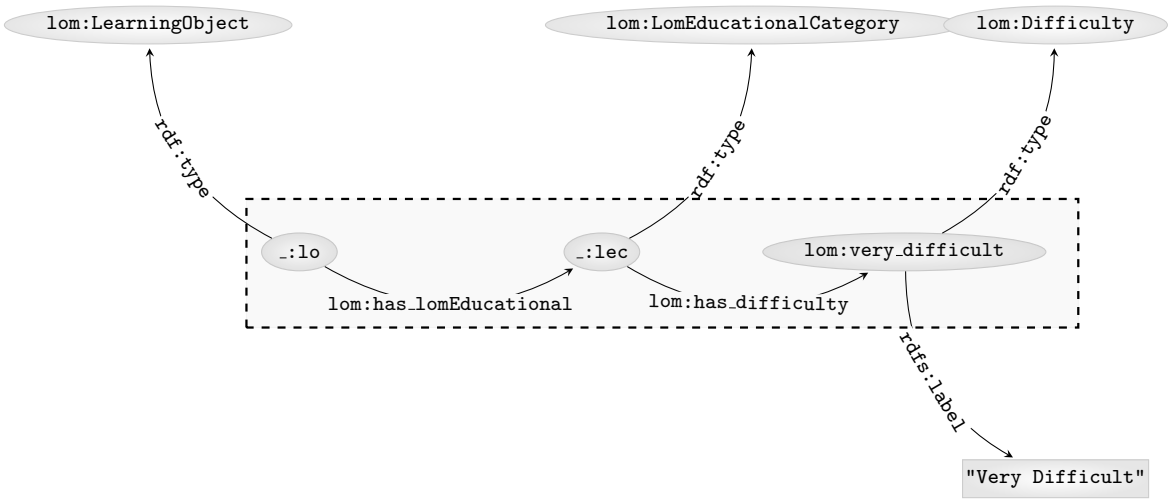


Figure 4.9. Representation of the query: *Very difficult documents*

The ontological elements related to the query are shown in figure 4.9.

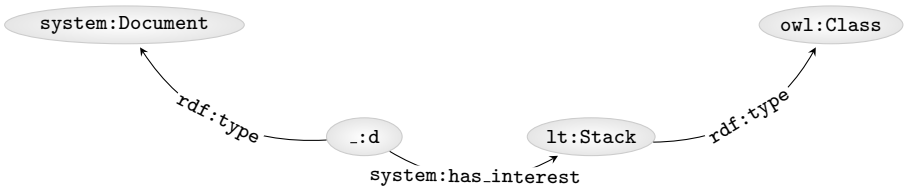
4.1.4.6 Query Extension

A query is an extension of another query when the resources it asks for concern a more general category of elements. The extension of a query is obtained starting from the first query. For this query there is a path in ontologies or knowledge bases. The extended query relies on another path created discovering the relations in the ontologies or knowledge bases among the elements of the first path.

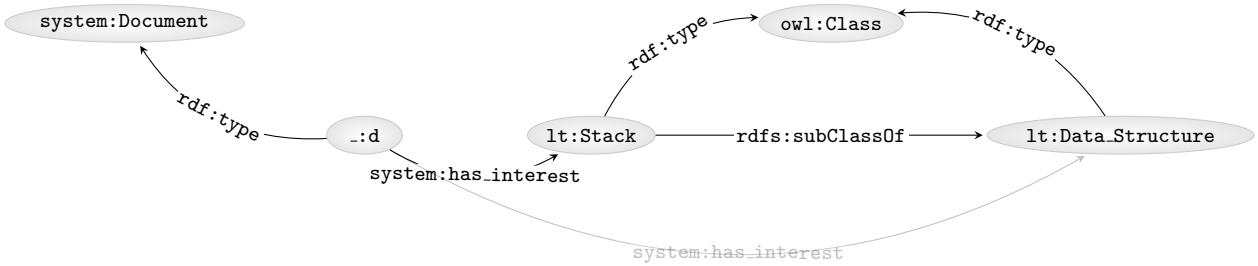
If the query is: *What are the documents about Stack?*

An extended query is: *What are the documents about Data Structure?*

The documents concerning *stack* also concern *data structure*. The concept *lt:Data_Structure* subsumes the concept *lt:Stack* (both concepts are part of the ontology concerning the *Theory of Languages*, denoted by *lt*).



(a) Knowledge base associated to documents about Stack



(b) Knowledge base associated to documents about Data Structure

Figure 4.10. Extended Query building

4.1.4.7 Range Queries

Our system has not been designed to answer certain types of queries: e.g. *What are the documents written between 1930 and 1940*? This is so because our system depends on the structure of DHTs: DHTs only support exact matching queries and do not support non-trivial queries involving temporal and geographic information [94]. A resource cannot be described with an interval of values. A *range query* [94] is the process of retrieving information from an interval of values given with a lower and an upper boundary. There are several works that face the problem of range queries, [27] [77], highlighting the fact that more scalable solutions require to store a distributed indexing data structure in the P2P network differently than for linear queries.

In certain cases we face queries that have not been mentioned before. These queries are not considered because they are outside the scope of this work.

4.1.4.8 Queries Formal Representation

The query represented in figure 4.9 displays some Learning Objects. It can be represented by the SPARQL query shown in figure 4.11 on the left. The results of this query denote resources that have an RDF description, similar to that on figure 4.11, on the right. For describing such resources the blank nodes identifiers have no meaning, only the structure of the triples are significant.

Query representation in SPARQL

```
SELECT ?x WHERE {
?x rdf:type lom:LearningObject .
?x lom:hasEducational ?y .
?y rdf:type lom:LomEducationalCategory .
?y lom:hasDifficulty lom:very_difficult .
}
```

Resource description in RDF

```
_:lo rdf:type lom:LearningObject .
_:lo lom:hasEducational :_lec .
_:lec rdf:type lom:LomEducationalCategory .
_:lec lom:hasDifficulty lom:very_difficult .
```

Figure 4.11. Representations of the same resources

The structure of the Graph Pattern of the SPARQL query is identical to the structure of the Ground Graph Pattern of the RDF description. For this reason, the same process can be used for formulating a query and a matching description.

4.1.4.9 Boolean Operators

A simple query is a query like *"what are the resources that present criteria A"*. We can translate with *"what are the resources that are described with criteria A"* and finally with *"what are the resources that have the description A"*.

Let q_A be the query with criteria A, and d_A the description A. q_A allows to retrieve documents with d_A.

Let *Doc* be the set of all resource identifiers in the index.

$Res(q_A)$ is the result of the query q_A. It contains the documents with the description d_A.

$$Res(q_A) = \{x \in Doc,\ \texttt{x has description}\ d_A\} \tag{4.1}$$

$Res(q_B)$ is the result of the query q_B. It contains the documents with the description d_B.

$$Res(q_B) = \{x \in Doc,\ \texttt{x has description}\ d_B\} \tag{4.2}$$

A complex query refers to a logical expression combining elementary criteria and boolean operators. We translate queries like *"what are the resources that present criteria A and criteria B"* with *"what are the resources that are described with criteria A and criteria B"* and finally with *"what are the resources that have descriptions A and description B"*. For example a query like *"what are the documents about Grammar that are Very Difficult"* can have answers because some resources have been described as *Documents about Grammar* and as *Documents Very Difficult*. They have been published with the description *Documents about Grammar* and with the description *Documents Very Difficult*.

$q_{A\&\&B}$ is the query with criteria A and criteria B. $Res(q_{A\&\&B})$ is the result of the query $q_{A\&\&B}$.

$$Res(q_{A\&\&B}) = Res(q_A) \cap Res(q_B) \tag{4.3}$$

The result of the complex query $q_{A\&\&B}$ requires two queries q_A and q_B and is the intersection of their results.

$$Res(q_{A\|B}) = Res(q_A) \cup Res(q_B) \tag{4.4}$$

The result of the complex query $q_{A\|B}$ requires two queries q_A and q_B and is the union of their results.

We have chosen not to implement the *not operator*.

$$Res(q_{\neg A}) = Doc \backslash Res(q_A) \tag{4.5}$$

Doc is the set of all resource identifiers in the index and q_A is the set of documents with the description d_A. The result of the complex query $q_{\neg A}$ requires two queries. The result of the complex query is the set-theoretic difference of their results. This set of resources is heterogeneous because it contains all the resources that do not have a description d_A. These kind of queries are not useful in a system dedicated to give affirmative responses.

OWL 2 [48] allows a kind of negation through the *NegativeObjectPropertyAssertion* and the *NegativeDataPropertyAssertion*. We can state that two individuals are not connected by a property, like in the following example that says that Luca is not Matteo's father:

$$NegativeObjectPropertyAssertion(:hasFather : Matteo : Luca) \tag{4.6}$$

Likewise, we can state that Luca's weight is not 80:

$$NegativeDataPropertyAssertion(:hasWeight : Luca"80"\hat{}\hat{}xsd : integer) \tag{4.7}$$

These features do not solve the issue of the *not operator* because they allow to create resource descriptions only for well identified triples, while nothing can be specified as negative description for the same resources.

4.2 Resources Description

4.2.1 Introduction

In our approach, resources are managed in the same way in both private and shared memories and have to be described by users. The same system of description is

used in both memories. We consider semantic descriptions built manually from the elements extracted from ontologies. Semantic descriptions are directly used for creating index entries. Semantic indexing has been designed so that it can be used to answer different types of queries presented in the previous section (see 4.1.4).

A resource can be described in different ways, and can be an answer for several queries. We have introduced two categories of queries, *Resource Query Type* and *Content Query Type*, depending on the elements concerning the resources themselves or more directly their content. The System Ontology added to domain ontologies allows us to unify resource descriptions in only one model. The reason for that is that resources have to be considered as instances of specific concepts of domain ontologies, or instances of the concept Document of the System Ontology. The difficulty is to present to users who are not aware of the knowledge representation, a navigational system that is able to guide the users in building resources descriptions.

The first element to be identified is the *Resource Type*, i.e. the initial concept that the user must select for denoting the current resource. Step by step, the process follows a path in one or two ontologies and stops when the user considers that the end of the path is reached or because the path cannot go on. The choice of properties and individuals during this process determines all elements contained in the final description. The system is able to build closed descriptions i.e. when all their elements are available in the ontologies. In some cases, the user is allowed to add the value of a property (the user can enter a keyword manually) and so the system completes an open description.

4.2.2 Sequence of Properties

Let's consider a sequence of two triples, (a,R,b) and (b,S,c), depicted in figure 4.12 as an RDF graph where the object of the first triple is the subject of the second one. We may group this sequence obtaining the triple (a,RφS,c), where RφS is the property result of the combination between R and S.

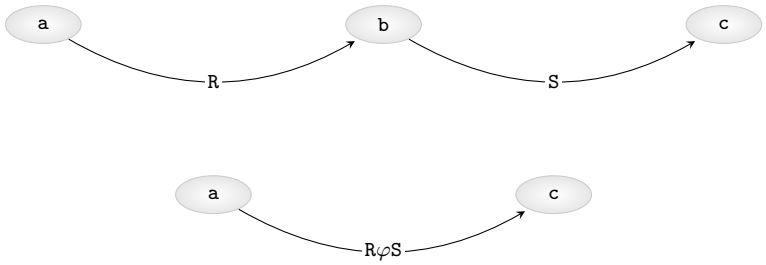


Figure 4.12. Sequence of triples

Definition 4.3. *(Operator followedBy)*

Given two properties R and S, RφS (R followedBy S) is a new property meaning that the relation R is applied first and then the relation S.

Let $\mathcal{I} = (\Delta^{\mathcal{I}},(\cdot)^{\mathcal{I}})$ (defined in section 2.2.1) be an interpretation. The semantics of the operator φ is given by:

$(R\varphi S)^{\mathcal{I}} = \{(x,y) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} | \exists z \in \Delta^{\mathcal{I}}, (x,z) \in R^{\mathcal{I}} \wedge (z,y) \in S^{\mathcal{I}}\}$

The definition is iterative and can be applied to a sequence of n properties pairwise.

4.2.3 Description Tree

When creating a description, the system has to consider a small knowledge base that results from the steps followed. This knowledge base can be represented as a tree, called *Description Tree* (hereafter *tree*).

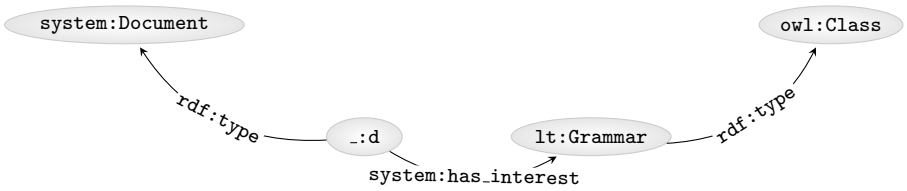
Definition 4.4. *(Description Tree)*

A Description Tree is an RDF graph. The root is always a blank node representing the resource referred by the description. Other nodes are also blank nodes, except nodes on the last level and the leaves, that are nodes identified by URIs or literals.

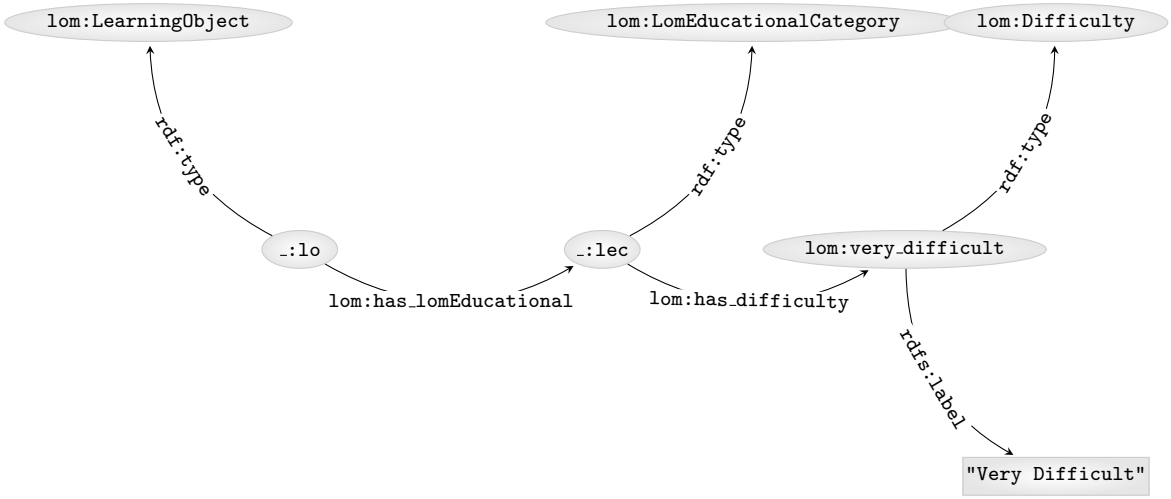
Let's consider the description: *documents about Grammar and Very Difficult*. The two components of the description (about *Grammar* and *Very Difficult*) refer to two different ontologies and are built separately. The knowledge base created by the system, resulting from the description steps can be represented by two trees (see figures 4.13(a) and 4.13(b)).

However, the logical description is unique and the resource can be represented as an instance of two concepts: *Document* in the *System Ontology* and *LearningObject* in the *LOM*.

Nodes *_:d* and *_:lo* have to be merged because they refer the same resource(see _:id_document in figure 4.14).



(a) Tree associated to documents about Grammar starting from _d



(b) Tree associated to documents Very Difficult starting from _lo

Figure 4.13. The query is considered split in two

Figure 4.14 shows the tree associated with the resource starting from the root *_:id_document*.

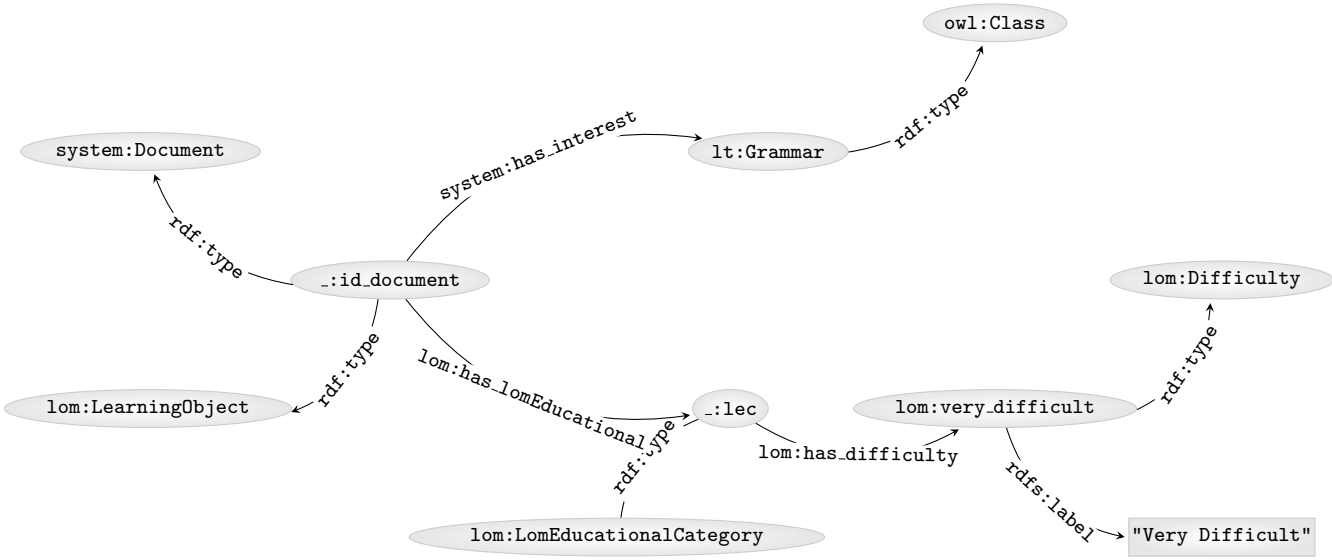


Figure 4.14. Tree associated to documents about Grammar, Very Difficult

4.2.4 Simple Description

Within a tree, the root denotes the selected resource and is represented as a blank node. The other nodes can be individuals or concepts in ontologies. Leaves can be literals. The arcs are properties that connect the nodes. Inside the tree, a path describes the backbone of the description.

Definition 4.5. *(Simple Description)*
A Simple Description is a Description Tree where the root has one and only one child (except those related to the property *rdf:type*). The tree in figure 4.15 follows n steps depending on n properties starting from the root. At each intermediate step, the element in the range of the property is in the domain of the following property and is represented by a blank node. A Simple Description is the semantic description of the resource.

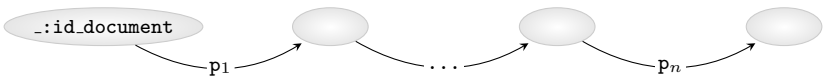


Figure 4.15. A Simple Description

The n properties may be assimilated to a sequence of properties. Combining the properties results in a compacted graph where the blank nodes are removed (see section 4.3 for blank node removing and the format of keys). A simple description

SDes may be associated to more than one resource. It is equivalent to a boolean predicate, p_{SDes}, on resources (res). The value of the predicate is *true* for each resource represented by *SDes* (having the same description).

$$p_{SDes}(res) = true, \texttt{ iff res is described by SDes}$$

A query based on a simple description Q_{SDes} allows one to retrieve the resources described by $SDes$.

The results of the query are these resources.

$$Result(Q_{SDes}) = \{res, p_{SDes}(res) = true\}$$

4.2.5 Complex Description

Definition 4.6. *(Complex Description)*

A Complex Description is a Description Tree where the root has more than one child. The tree is the merging of the n simple descriptions combined with the AND boolean operator, where n is the number of children (except those related to the *rdf:type* property) of the root. A Complex Description *CDes* is defined by the union of simple descriptions:

$$CDes = SDes_1 \vee SDes_2 \vee ... \vee SDes_n$$

A Complex Description contains several paths. Each path starts from the root and relates a Simple Description *SDes*.

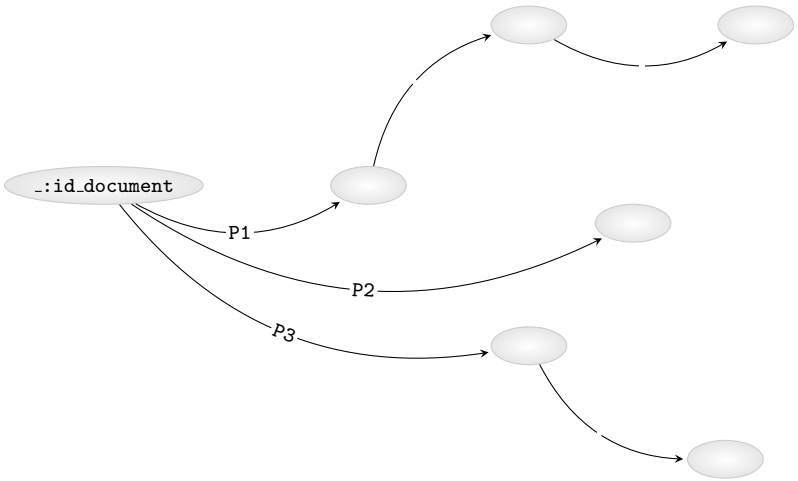


Figure 4.16. Tree associated to a complex description

In figure 4.16 a complex description is represented by a unique tree. Three paths start from root and follow the three properties P_1, P_2 and P_3.

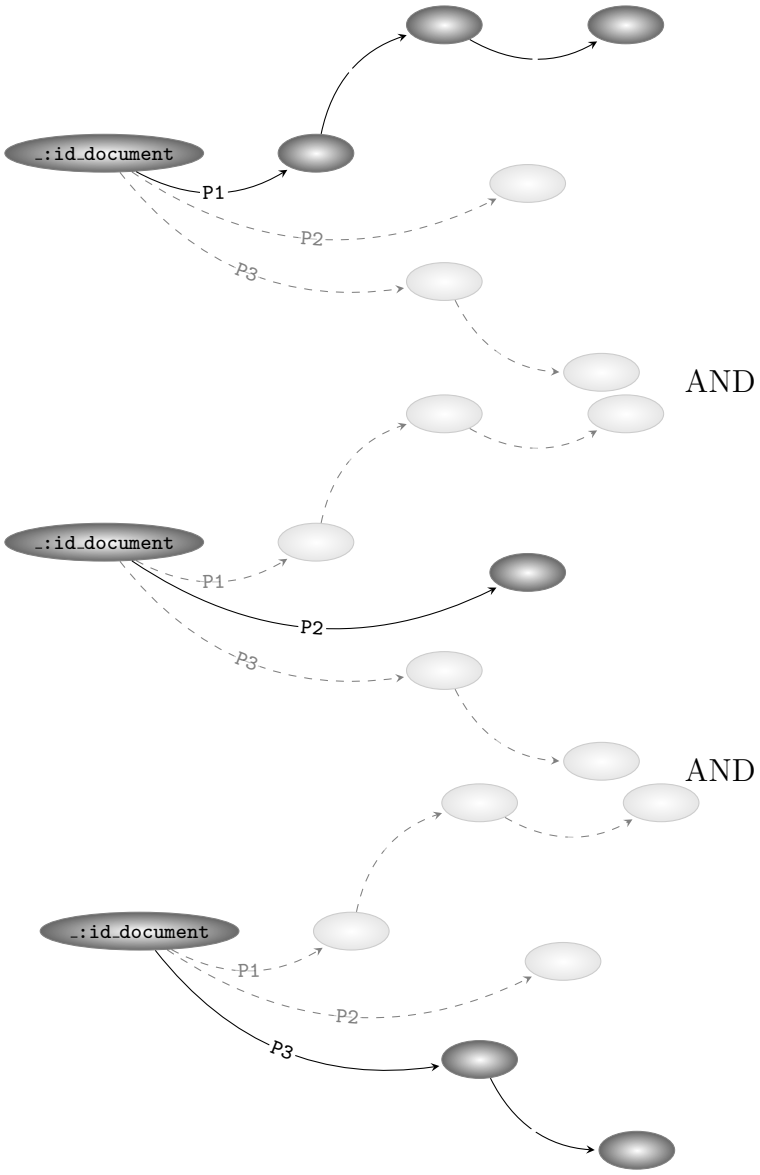


Figure 4.17. Each path corresponds to a description

Considering n different paths addressing n resources, the Complex Description $CDes$ is the union of the Simple Descriptions:

$$CDes = \bigcup_{i=1,n} SDes_i$$

For each resource res represented by $CDes$, the value of the predicate p_{CDes} is true and:

$$p_{SDes_i}(res) = true, \forall i = 1,n$$

A query requesting for resources having a complex description will be considered as a set of elementary queries about resources having a simple description. The result of the query will be the intersection of the elementary query results.

$$Result(Q_{CDes}) = \bigcap_{i=1,n} Result(Q_{SDes_i})$$

4.3 Creation of Keys

4.3.1 Introduction

In the system, a Semantic Description is associated with one or more resources. The resources are catalogued within the Semantic Index. We call *index* the Semantic Index and *indexing* the Semantic indexing. The index is composed of entries that are pairs of data (key,value). The key is generated from the semantic description, that in its turn is created from the path followed by the user for describing the resource. The same indexing is used for both personal and shared memories (see chapter 5 for details).

The *distributed index*, that concerns the shared memory, is allocated in the DHT of the P2P network. Each node of the network contains a portion of the whole DHT. The value of an entry is a list of elements containing the access point to the resource addressed by the key. A resource is accessed either through its URL (it is accessible when the URL is reachable), or through its entire data content (in this way the access to this resources is completely decentralized and the resources are available while the DHT is running). See figure 4.18. Some resources require their content be inserted into the DHT. They concern either the core elements of the System, e.g. the ontologies used to create the keys of indexing, or the elements bound to the community, e.g. the Wiki of the Community, Personal Notes (details in section 4.6).

The *local index*, that concerns the personal memory, is stored in a local archive within the peer. Resources are always files. The value of an entry is the reference of the resource in the local file system.

Keys are used in the system for the publication and retrieval of resources. The publication of a resource may lead to the creation of a new key and a new entry in the index, or the adding of the resource reference into an existing entry when the key used for describing the resource already occurs in the index. The retrieval allows to find in the index the resources that correspond to a research key.

A key used in the index is a representation of the semantic description of a resource and is written in a language based on RDF.

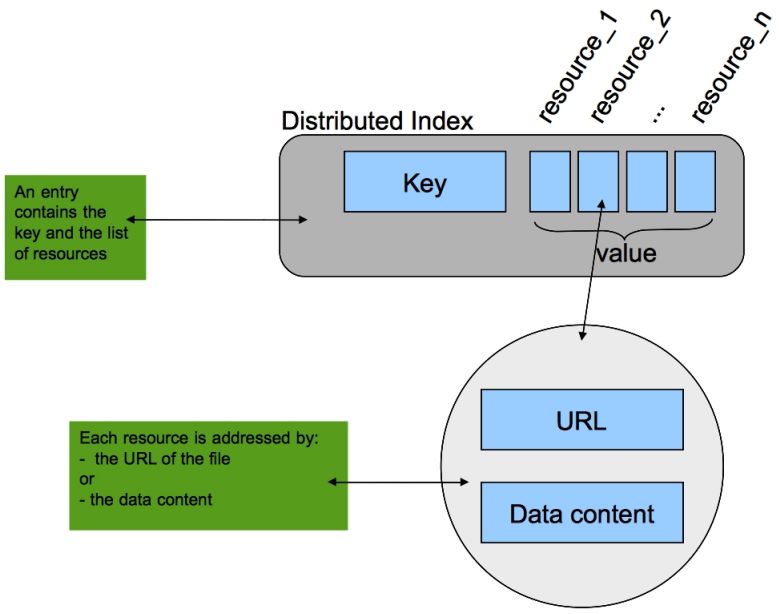


Figure 4.18. An entry of the distributed index

4.3.2 Description Representation

Keys are generated considering the semantic description system (detailed in section 4.2), i.e. considering the two possible types of queries included in our system. For creating a key, it is necessary to go through the *Description Tree* associated to the resource. A *Simple Description* generates a single key; multiple keys are produced from a *Complex Description*.

The two approaches we have considered for defining the semantic indexing of resources have been unified thanks to the creation of the System Ontology (see section 4.1.2.3 for details).

Let's consider as example the resources that are described as very difficult, associated to the query *What are the Very Difficult documents?* It is a case of *Resource Query Type* (see section 4.1.4.2).

This sample requires an ontology concerning the domain of *Learning Object* that contains the concept of *Difficulty*. The namespace name of this ontology and the nature of its concepts are enough to ensure that it represents the right knowledge domain. The user knows the *Learning Object Model* ontology [39] (denoted by *lom*). Following the concepts and relations of this ontology, the resources have to be

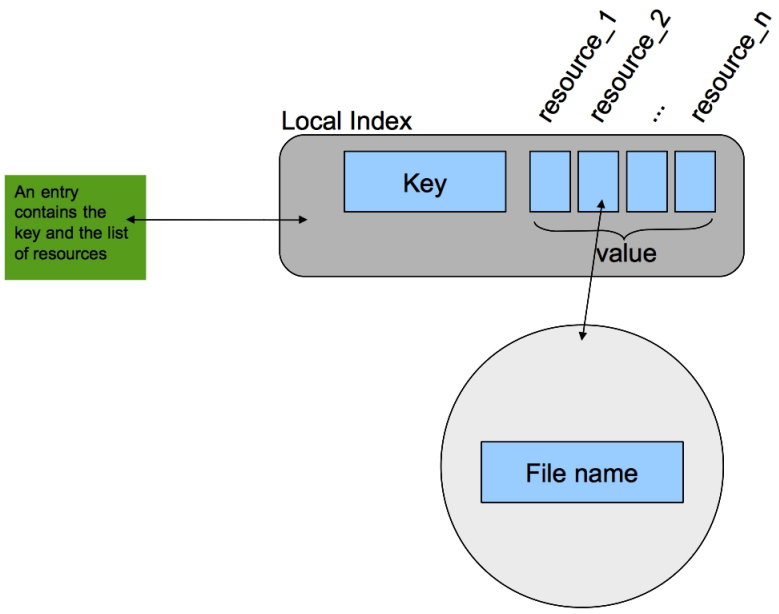


Figure 4.19. An entry of the local index

considered as *learning objects* that have an *educational category* whose *difficulty level* is *very difficult*. The semantic description is an RDF graph (the *Description Tree*, as described in section 4.2.3) that contains blank nodes useless for indexing because they do not contain semantic information necessary for describing a resource. The description contains the following triples:

```
_:lo rdf:type lom:LearningObject .
_:lo lom:hasEducational :_lec .
_:lec rdf:type lom:LomEducationalCategory .
_:lec lom:hasDifficulty lom:very_difficult .
```

The N3 notation [12] synthesizes the description as follows:

```
[ a lom:LearningObject ] lom:hasLomEducational
[a lom:LomEducationalCategory ;
   lom:hasDifficulty lom:veryDifficult .]
```

Such a description shows the triples that hide the unnecessary blank nodes. Moreover, applying the combination of properties (operator $\varphi : followedBy$) the description can be represented by:

```
[ a lom:LearningObject ]
lom:hasLomEducational φ lom:hasDifficulty lom:veryDifficult .
```

Then, a key used for indexing a resource is based on the above representation. The format of the key is the following:

```
Key_1:
   {rdf:type,lom:LearningObject}
   {lom:hasLomEducational}
   {lom:hasDifficulty,lom:veryDifficult}
```

The first line represents the type of the resource, the second line is the first considered property, and the third line is the second considered property and its value. If more than two properties are combined by the *followedBy* operator, the key will be composed of more lines similar to the second one.

The second example concerns *Content Query Type* (see section 4.1.4.3). It occurs when a user does not consider an ontology where a concept can represent the type of the resources. In this case, the user only assumes that the resources refer to a subject. For example, a teacher can say that a resource is about deterministic finite automaton (*dfa*). We only consider that the topic of the resource has to be associated to a concept of an ontology. It is necessary to use the *system:Document* concept to represent the type of such a resource and the *system:hasInterest* property to link the resource to a concept of another ontology that qualifies its content. The description contains the following triples:

```
_:d rdf:type system:Document .
_:d system:hasInterest lt:dfa .
```

In the last triples *lt* denotes an ontology about Theory of Languages. The N3 notation synthesizes the description as follows:

```
[a system:Document] system:hasInterest lt:dfa
```

The format of the key is the following:

```
Key_2:
   {rdf:type,system:Document}
   {system:hasInterest, lt:dfa}
```

In the latter case, it is not necessary to use the *followedBy* operator for simplifying the path of the description.

To produce the final format of the keys, the namespaces prefixes are substituted by the related namespaces names (URIs) and hash coding is applied.

4.3.3 Context Extension

4.3.3.1 Definition

To create a key, we distinguish between the publication and retrieval contexts. They do not involve the same conditions and circumstances.

Definition 4.7. *(Publication Context)*
We call Publication Context the description supplied for publishing a resource.

The publication context is used for creating the key associated to the resource in the distributed index.

Definition 4.8. *(Retrieval Context)*
We call Retrieval Context the description supplied for searching resources.

A retrieval context is the description of a required resource and must correspond to a publication context. For retrieving a resource a key must be supplied and must be equal to the key created from a publication context. In a boolean index, as DHT, keys used for retrieving resources must be equal to keys used for publishing. However people should be able to find a resource with other characteristics than those exactly used for publishing. Publication and retrieval contexts are different, but may lead to the same resources. Considering the shared memory, an important issue to take into account is the number of queries that have to be launched through the network. It must be minimal in order to minimize the access time to the resources.

It is necessary to consider an extension process in order to retrieve a resource in case of different requests. We can tackle this issue either at publishing time, thanks to a description extension, or at retrieval time thanks to a query extension. The last option involves producing several queries and in the case of resources retrieved from the network, would be too time consuming. It is thus necessary to prepare a description extension used when a new resource is published. There is no problem with time at this stage of the process. The context extension produces a *Complex Description* (see section 4.2.5) obtained combining the *Simple Description* supplied by the resource provider with others generated by the system. The resource is then published with different keys, one for each of the *Simple Descriptions*.

In this work we consider three cases of context extension.

4.3.3.2 Subsumption

Let's consider the resource _:d about the concept of Stack (found in the ontology *Theory of Language*).

```
_:d
   a system:Document ;
   system:hasInterest lt:Stack .
```

In this ontology, the concept Stack has a super-concept: Data_Structure. We consider that any request of resources concerning Data_Structure should also return resources concerning Stack because Stack is a specialization of Data_Structure.

In this case of publication context extension of the _:d resource, we consider the generalization of the Stack concept. As the content of _:d is about Stack, it is also concerning Data_Structure type. So, we must identify two keys:

```
Key_initial:
   {rdf:type,system:Document}
   {system:hasInterest, lt:Stack}
Key_extended:
   {rdf:type,system:Document}
   {system:hasInterest, lt:Data_Structure}
```

The process of the subsumption context extension could involve several steps of generalization in case of more levels in the hierarchy of concepts. However, we choose to limit to one level in order to avoid too broad generalizations of requests.

4.3.3.3 Facet Extension

We consider the resource described as

```
[ a lom:LearningObject ] lom:hasLomEducational
[a lom:LomEducationalCategory ;
   lom:hasDifficulty lom:veryDifficult .]
```

that brings to create the key:

```
Key_initial:
   {rdf:type,lom:LearningObject}
   {lom:hasLomEducational}
   {lom:hasDifficulty,lom:veryDifficult}
```

However, it is interesting to notice that such resource could be found from other queries. For example, a user may be interested in learning objects where the difficulty level has been defined, no matter its value. In this case, the retrieval context is a description like the following, where <value> represents any instance of the concept range of the property lom:hasDifficulty.

```
_:lo
   rdf:type lom:LearningObject ;
lom:hasLomEducational _:lec .
_:lec
   lom:hasDifficulty <value> .
```

It is not possible to create a key for each possible value. For representing this idea, we decided to apply an extension of the key formalism, where the value of the

last property is omitted. The corresponding key is:

Key_extended:
{rdf:type,lom:LearningObject}
{lom:hasLomEducational}
{lom:hasDifficulty}

As a result, the resource will be published with two keys:

Key_initial:
{rdf:type,lom:LearningObject}
{lom:hasLomEducational}
{lom:hasDifficulty,lom:veryDifficult}
Key_extended:
{rdf:type,lom:LearningObject}
{lom:hasLomEducational}
{lom:hasDifficulty}

4.3.3.4 Category Extension

We consider resources *about the author Chomsky*. The case is included in a *Content Query Type* and we find an ontology on *Theory of Languages* where the element *Chomsky* is defined as an individual of the concept *Author*.

The resource is described as:

[a system:Document] system:hasInterest lt:chomsky

The key consequence to such a description is

Key_initial:
{rdf:type,system:Document}
{system:hasInterest, lt:chomsky}

This description can be expanded because its meaning may be intended also as referring to a resource *about an author* in the same domain. In this case the meaning is that a resource whose content is about a particular author, is also about the concept of Author. In this way we consider possible to expand the description to the category of the individual. The description, as a consequence of the extensions is:

[a system:Document] system:hasInterest lt:chomsky
[a system:Document] system:hasInterest lt:Author

We can identify two keys:

```
Key_initial:
   {rdf:type,system:Document}
   {system:hasInterest, lt:chomsky}
Key_extended:
   {rdf:type,system:Document}
   {system:hasInterest, lt:Author}
```

4.3.4 Cases of Indexing

During this work we have seen, through various examples, some cases of indexing. In this section we will see all the cases of indexing taken into consideration in our system. They constitute an integration of the examples already provided in this work. Moreover, they explain some cases not yet considered. In each case we tackle the process of description and the creation of the keys of indexing. In some cases it is necessary to perform slight adaptations of the process of creation of the keys. In the event that the ontologies do not provide evident suitable elements for the description, the user is allowed to add manually in the description a string that represents the value of a property. This possibility is very important because it allows to deal with knowledge bases where some interesting individuals are missing. Some cases of indexing lead to a new kind of context extension. We also considered the case where no ontologies are available for describing a resource. We thought that it was interesting to let the user to associate keywords with a resource even it represents an exception from a semantic point of view.

4.3.4.1 Indexing on the Resource Type

The ontology denoted by *tg* (in the e-government domain) contains the concept *tg:Z-Document* that represents a report or a legal text. The following description

```
[a tg:Z-Document]
```

represents a resource of type *tg:Z-Document*.

We do not accept the indexing that is based only on the type of a resource, like in the previous example, because the result of the corresponding query would contain too many elements and would not be discriminating enough.

We can also describe a resource of type *tg:Z-Document* giving the *tg:Z-Program* to which it refers:

```
[a tg:Z-Document] tg:hasProgram tg:Z-Program
```

If we had accepted the indexing on the type of a resource we should have extended any other description that gives also a property to the resource. Indeed a resource having a certain type T and which is specified, has also to be considered as a resource

of type T.

4.3.4.2 Indexing on a Concept

This case of indexing relates a resource treating a particular topic. The description of such a resource is given by elements that concern its content. For instance we consider a *Resource treating of Grammar*. The elements of descriptions belong to the *System Ontology* and to an ontology of domain which provides the concept expressing the content of the resource.

The description of the resource contains the following triples, where <C> represents any concept:

```
_:d rdf:type system:Document .
_:d system:hasInterest <C> .
```

The format of the key is:

```
Key:
   {rdf:type,system:Document}
   {system:hasInterest, <C>}
```

In the case of subsumption extension, the indexing case remains the same because a super-concept of a concept is also a concept.

4.3.4.3 Indexing on a Property

We can consider a resource about the life cycle of a learning object. According to the LOM Ontology this is translated by the fact that a learning object has a life cycle. The description of the resource contains the following triples:

```
_:d rdf:type system:Document .
_:d system:hasInterest lom:hasLifeCycle .
```

This is a specific case of a description like the following, where <P> represents any property:

```
_:d rdf:type system:Document .
_:d system:hasInterest <P> .
```

The format of the key is:

```
Key:
   {rdf:type,system:Document}
   {system:hasInterest, <P>}
```

We consider the subsumpion of properties as well as the subsumption of concepts and the indexing case remains the same.

4.3.4.4 Indexing on an Individual

Let's consider a resource treating of *lt:chomsky* which is an instance of the concept *lt:Author*. The description of the resource contains the following triples:

```
_:d rdf:type system:Document .
_:d system:hasInterest lt:chomsky .
```

It is a particular case on indexing on an individual <I>.

```
_:d rdf:type system:Document .
_:d system:hasInterest <I> .
```

The format of the key is:

```
Key:
   {rdf:type,system:Document}
   {system:hasInterest, <I>}
```

The application of the category extension to this case leads to the indexing of a concept:

```
_:d rdf:type system:Document .
_:d system:hasInterest lt:Author .
```

Consequently, this case generates two keys of the form:

```
Key_initial:
   {rdf:type,system:Document}
   {system:hasInterest, <I>}

Key_extended:
   {rdf:type,system:Document}
   {system:hasInterest, <C>}
```

4.3.4.5 Indexing on a Keyword

It could happen that the user does not find any element, within an ontology, useful enough for describing the content of a resource. Therefore it is necessary to allow the user to insert a keyword manually. The System Ontology provides the property *system:hasKeyword* defined for this purpose. Its range defines the type of the keyword as xsd:string. For example let's consider a resource about *Medieval Italy*. Without any ontology about this topic, we could at least describe the resource with the following triples:

```
_:d rdf:type system:Document .
_:d system:hasKeyword "Medieval Italy"^^xsd:string .
```

The general description of this case of indexing is:

```
_:d rdf:type system:Document .
_:d system:hasKeyword <k>^^xsd:string .
```

The format of the key is:

```
Key:
   {rdf:type,system:Document}
   {system:hasKeyword, <k>}
```

The description provider should follow the guidelines for writing keywords, otherwise it could be impossible for other users to find resources described in such a way.

4.3.4.6 Indexing on a Virtual Individual

In the field of the Theory of Languages, *Jeffrey D. Ullman* is a famous author. Considering the ontology denoted by *lt* we cannot find any individual of the concept lt:Author referring to *Jeffrey D. Ullman*. Moreover, we cannot find in this ontology any strings that contain the word *Jeffrey D. Ullman*. The system gives the user a possibility to consider a virtual individual of the concept lt:Author and to specify its name by inserting manually the value of the property lt:hasName. The virtual individual is used in the description of the resource about the author *Jeffrey D. Ullman*. The description of the resource contains the following triples:

```
_:d rdf:type system:Document .
_:d system:hasInterest _:a .
_:a rdf:type lt:Author .
_:a lt:hasName "Jeffrey D. Ullman"^^xsd:string .
```

If we apply the mechanism of key creation presented in 4.3.2, we obtain the following result:

```
Key:
   {rdf:type,system:Document}
   {system:hasInterest}
   {lt:hasName "Jeffrey D. Ullman"^^xsd:string}
```

This representation has some drawbacks:

- there is no way to determine the type of the virtual individual;
- the property lt:hasName could refer to a concept other than lt:Author

In the case of a real individual, such as lom:very_difficult, it is possible to determine its type lom:Difficulty through a query on the ontology. As a consequence, we consider it essential to maintain the type of the virtual individual in the key:

Key:
{rdf:type,system:Document}
{system:hasInterest,lt:Author}
{lt:hasName,"Jeffrey D. Ullman"}

This is a particular case of a generic key:

Key:
{rdf:type,system:Document}
{system:hasInterest,<C>}
{<P>,<k>}

where <C> is the type of the virtual individual, <P> is the property which domain contains the type of the virtual individual, and <k> is the ^^xsd:string value the of the property <P> the user inserts manually.

As this case is similar to the case 4.3.4.4, it is necessary to apply the category extension and to create another key whose the generic shape is the following:

Key_extended:
{rdf:type,system:Document}
{system:hasInterest, <C>}

Moreover, we can consider a new type of extension called "Keyword Extension" where the string added as value of a property in the description of a resource, is also a keyword that can refer it. This case of extension gives another key:

Key_extended:
{rdf:type,system:Document}
{system:hasKeyword,<k>}

4.3.4.7 Iterative Indexing

This case of indexing refers to the resource and not to its content. The description of such resource is given by elements that concern the type of the resource and some of its properties. The elements of description are constituted by a sequence of properties. For instance we consider *a resource describing an Educational Program applied in France*. This resource can be described using the ontology GeoSkills, denoted by *geo*. We create a description containing the following triples:

_:d rdf:type geo:EducationalProgram .
_:d geo:inEducationalRegion geo:France .

In this example there is only one step (*geo:inEducationalRegion*) between the resource (*_:d*) and the individual (*geo:France*) that ends the description path. The corresponding keys are:

Key_initial:
{rdf:type, geo:EducationalProgram}
{geo:inEducationalRegion, geo:France}

Key_extended:
{rdf:type, geo:EducationalProgram}
{geo:inEducationalRegion}

The *Key_extended* is obtained applying the facet extension.

The corresponding keys are a particular case of the general one:

Key_initial:
{rdf:type, <C>}
{<R>, <I>}

Key_extended:
{rdf:type, <C>}
{<R>}

Where <C> is the type of the resource, <R> a relation and <I> an individual.

The example of section 4.3.3.3 describes *resources very difficult*. This is a case with two steps between the resource itself and the individual <I> that ends the path of the description.

Now we can consider a *resources having the University of Compiègne (UTC) as contributor*. Using the *lom* ontology the corresponding description involves the following triples:

_:d rdf:type lom:LearningObject .
_:d lom:hasLomLifeCycle _:b_1 .
_:b_1 lom:hasContributeElement _:b_2 .
_:b_2 lom:hasEntity lom:utc .

The keys created from the description are:

Key_initial:
{rdf:type, lom:LearningObject}
{lom:hasLomLifeCycle}
{lom:hasContributeElement}
{lom:hasEntity, lom:utc}

Key_extended:
{rdf:type, lom:LearningObject}
{lom:hasLomLifeCycle}
{lom:hasContributeElement}
{lom:hasEntity}

In this example there are three steps from the resource *_:d* and the individual *lom:utc* ending the description.

The generic case of an iterative indexing implies n steps, represented by $n \geq 1$ relations. A resource description contains the following triples:

_:d rdf:type <C>.
_:d <R_1> _:b_1 .
_:b_1 <R_2> _:b_2 .
{...}
_:b_{n-1} <R_n> <I> .

The general form of the keys is the following:

Key_initial:
{rdf:type, <C>}
{<R_1>}
{...}
{<R_n>, <I>}

Key_extended:
{rdf:type, <C>}
{<R_1>}
{...}
{<R_n>}

4.3.4.8 Iterative Indexing Involving a Virtual Individual

Sometimes the ending individual is not defined within the ontology containing the type of the resource. For that reason, our system gives the possibility to consider a virtual individual and to insert it in the description giving one of its properties. In the last example, the individual *lom:utc* could not be defined in the *lom* ontology.

It is enough to give the type and the value of the attribute *lom:name* to qualify the added virtual individual. The description of the resource would be:

_:d rdf:type lom:LearningObject .
_:d lom:hasLomLifeCycle _:b_1 .
_:b_1 lom:hasContributeElement _:b_2 .
_:b_2 lom:hasEntity _:vi .
_:vi rdf:type lom:Organization .
_:vi lom:name "University of Compiègne"^^xsd:string .

The keys generated from the latter description are:

Key_initial:
{rdf:type, lom:LearningObject}
{lom:hasLomLifeCycle}
{lom:hasContributeElement}
{lom:hasEntity, lom:Organization}
{lom:name, "University of Compiègne"^^xsd:string}

Key_extended_1:
{rdf:type, lom:LearningObject}
{lom:hasLomLifeCycle}
{lom:hasContributeElement}
{lom:hasEntity}

Key_extended_2:
{rdf:type, system:Document}
{system:hasKeyword, "University of Compiègne"}

The resources addressed by this type of index are described by the following triples expressed in the general form:

_:d rdf:type <C>.
_:d <R_1> _:b_1 .
_:b_1 <R_2> _:b_2 .
{...}
_:b_{n-1} <R_n> _:vi .
_:vi rdf:type <T>.
_:vi <P> <k> .

The general form of the keys is:

Key_initial:
{rdf:type, <C>}
{<R_1>}
{...}
{<R_n>, <T>}
{<P>, <k>}

Key_extended_1:
{rdf:type, <C>}
{<R_1>}
{...}
{<R_n>}

Key_extended_2:
{rdf:type, system:Document}
{system:hasKeyword, <k>}

This possibility leads to the keyword extension, where the keyword inserted by the resource provider as value of the property giving information on the virtual individual. This extension was presented in a case that refers to the content indexing of a resource. In this case the extension starts from a resource indexing case and lost this characteristic. It seems to us more interesting to give access to the resource from keyword, even if the original intention of the resource provider is not maintained.

So far we have considered the description of resources referring to their content or to the resources themselves. The second type involves iterative indexing. We have chosen this distinction in order to have clearer exposition. However it is possible to find a property in an ontology that refers to the content of a resource, for instance in the *foaf* ontology the property *foaf:primaryTopic*. So, the limit between both cases of indexing is not so strict. Even if there is this possibility, for indexing a resource about its content, we suggest to follow the processes shown in sections 4.3.4.2, 4.3.4.3, 4.3.4.4 and 4.3.4.5, that present a wider set of possibilities.

4.4 Use of Ontologies

4.4.1 Ontological Elements

Our solution leaves open the choice of the ontologies required for the resource description. However, users of the community should share them, otherwise the discovery of the documents published in the network would be impossible. The ontologies are published in the network as other resources. The publishing of an ontology requires an index key and a small additional description (see figure 4.20). Such

description is additional information integrated in the value of the index entry when the ontology is published. It includes the application domain of the ontology, the set of *Entry Points*, and the URI identifying the namespace name of the ontology. The additional information is used by users when they discover the ontologies in the system and helps them for choosing the needed ontology.

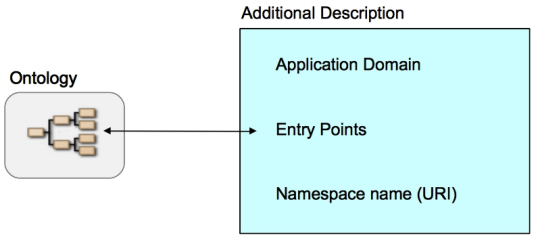


Figure 4.20. The description provided when an ontology is published

For example, the *lom* ontology concerns the domain of e-Learning, has as its *Entry Point* the concept *lom:LearningObject* and has as its namespace name the URI `http://www.owl-ontologies.com/2007/04/Jijel+UTC/lom.owl`; the *foaf* ontology describes links between people, has for *Entry Point* the concept *foaf:Document* and has for namespace name `http://xmlns.com/foaf/0.1/`

The resource provider is responsible for the choice of the ontology. The manual semantic indexing requires the selection of the ontologies used for building the indexing key. A key may contain several concepts that belong to one or two ontologies. Within the ontology, an element is completely defined by its unique URI[2] . It is enough to insert the URIs of ontological elements for characterizing a resource in the key of its indexing. Starting with this information, any software agent can discover the type of the element inside a key and can decide to build new queries if some queries do not give satisfying results.

[2]http://www.w3.org/Addressing/

4.4.2 The System Ontology

Resource description is based on the two models (described in section 4.1.4), defined as *Resource Query Type* and *Content Query Type*. To unify the two models, we have created the *System Ontology*. The basis is that resources have to be considered as instances of specific concepts of domain ontologies, or instances of the concept *Document* of the *System Ontology*.

Content Query Type aims at finding resources from elements of description that concern their content. Such type of queries concern the content of the resource. They ask for a resource about something. The resource provider means that the topic of the resource is about a concept defined in an ontology without any other specification. Users must discover the resource when they tell to the system they are interested in resources concerning this concept. The *System Ontology*, denoted by *system*, allows such descriptions. It contains the concept *system:Document* and the relation *system:hasInterest* for representing this case. Using the *System Ontology* and the ontology of Theory of Languages domain, denoted by *lt*, it is possible to describe a document concerning the concept of Automaton with:

```
[ a system:Document ]
system:hasInterest lt:Automaton.
```

The logical complexity of this representation is OWL Full. Inference in OWL Full is clearly undecidable as OWL Full does not include restrictions on the use of transitive properties which are required in order to maintain decidability. It is not a problem for us because we only consider the subsumption of concepts. Even if it is not useful for us, for maintaining the DL level of the description we could have substituted the concept lt:Automaton with an anonymous instance lt:_automaton. The correct meaning of the description would be: *a resource concerning any example of automata*.

Ontologies have to be indexed and published like any other resource. The system ontology is automatically published when the system starts (see section 4.6). A special key, using the *System Ontology*, is created for the publication and the discovery of the ontologies in the system. The concept *system:Ontology*, subconcept of *system:Document* is created for this purpose. An ontology used for indexing is published in the system under the description:

```
[a system:Ontology]
```

This description is created by the system. We can notice that this kind of description is not sufficient and it is not accepted by the system when a user wants to publish a resource (as detailed in section 4.3.4.1).

In the case that no ontology is available, the system ontology is also useful for associating some keywords with a resource.

In figure 4.21 we can see the hierarchy of concepts defined in the System Ontology.

The top concept is *system:Document*. It is used in the descriptions of *Content Query Type* for addressing the resource to be described.

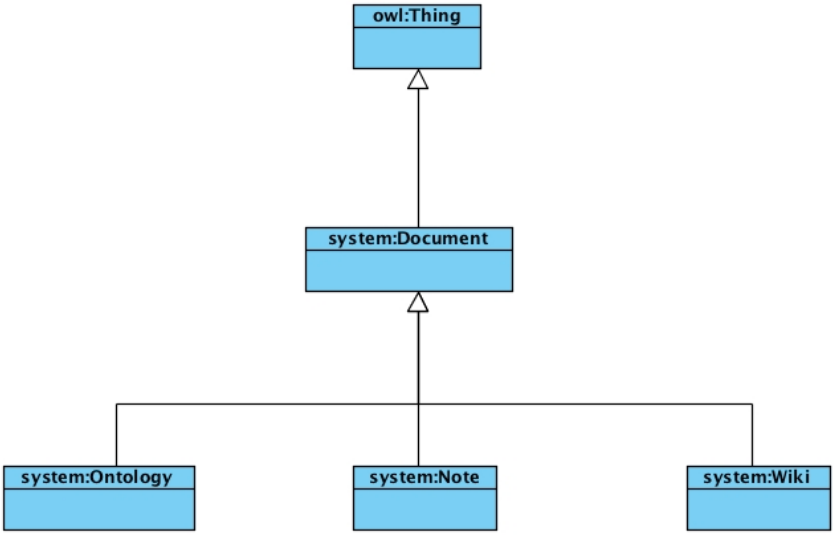


Figure 4.21. The concepts defined in the System Ontology

The *system:Document* concept is defined with the following OWL fragment:

system:Document

```
<owl:Class rdf:ID="Document">
   <rdfs:subClassOf rdf:resource="owl:Thing"/>
   <rdfs:label xml:lang="en">Document</rdfs:label>
   <rdfs:label xml:lang="fr">Document</rdfs:label>
   <rdfs:label xml:lang="it">Documento</rdfs:label>
   <rdfs:comment xml:lang="en">
     The type of a resource used when describing its content.
   </rdfs:comment>
</owl:Class>
```

The concept *system:Document* is super concept of three concepts *system:Ontology*, *system:Note system:Wiki*. These concepts have been created to represent the resources of the system (the resources part of the community, see section 4.6). They are used for describing resources on the *Resource Query Type*, for publishing the resources of the system. Through these concepts it is possible to create descriptions such as:

[a system:Ontology]
[a system:Note]
[a system:Wiki]

for publishing ontologies, system notes and the system wiki.
The following OWL fragments detail their definitions:

system:Ontology

```
<owl:Class rdf:ID="Ontology">
   <rdfs:label xml:lang="en">Ontology</rdfs:label>
   <rdfs:label xml:lang="fr">Ontologie</rdfs:label>
   <rdfs:label xml:lang="it">Ontologia</rdfs:label>
   <rdfs:comment xml:lang="en">The type of the ontologies used in the system
            for indexing the resources.
   </rdfs:comment>
   <rdfs:subClassOf rdf:resource="system:Document"/>
</owl:Class>
```

system:Note

```
<owl:Class rdf:ID="Note">
   <rdfs:label xml:lang="en">Note</rdfs:label>
   <rdfs:label xml:lang="fr">Note</rdfs:label>
   <rdfs:label xml:lang="it">Appunto</rdfs:label>
   <rdfs:comment xml:lang="en">The type of the notes used in the system by users
            as short written     reminders.
   </rdfs:comment>
   <rdfs:subClassOf rdf:resource="#Document"/>
</owl:Class>
```

system:Wiki

```
<owl:Class rdf:ID="Wiki">
   <rdfs:label xml:lang="en">Wiki</rdfs:label>
   <rdfs:label xml:lang="fr">Wiki</rdfs:label>
   <rdfs:label xml:lang="it">Wiki</rdfs:label>
   <rdfs:comment xml:lang="en">The type of the wiki of the system used by users
            for sharing editable content.
   <rdfs:subClassOf rdf:resource="#Document"/>
   </rdfs:comment>
</owl:Class>
```

The System Ontology defines the OWL object property *system:hasInterest* (see figure 4.22). Such property is necessary for describing relations in *Content Query Types*. The property links a *system:Document* with any resource represented by a Uniform Resource Identifier Reference (URI).

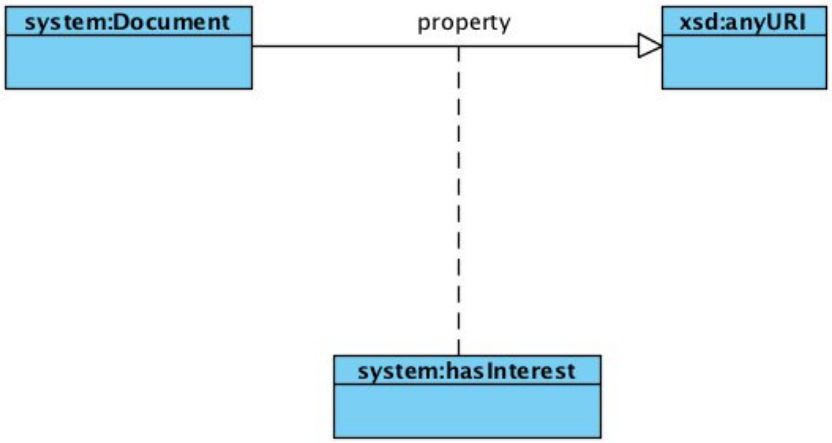


Figure 4.22. The property *system:hasInterest* defined in the System Ontology

The property *system:hasInterest* is defined in OWL as follows:

system:hasInterest

```
<owl:DatatypeProperty rdf:ID="hasInterest">
  <rdfs:label xml:lang="en">Has Interest</rdfs:label>
  <rdfs:label xml:lang="fr">A intérêt en</rdfs:label>
  <rdfs:label xml:lang="it">Fa riferimento a</rdfs:label>
  <rdfs:comment xml:lang="en">
    A property that links a Document to an ontological element that
   denotes its content.
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Document"/>
  <rdfs:range rdf:resource="xsd:anyURI"/>
</owl:DatatypeProperty>
```

In addition to the latter property, the System Ontology defines the OWL datatype property *system:hasKeyword* (see figure 4.23). Such property is necessary in *Content Query Types* for linking *system:Document* with an *xsd:string*. Usually the property is used to assert that the given keyword is contained within the resource.

The property *system:hasKeyword* is defined in OWL as follows:

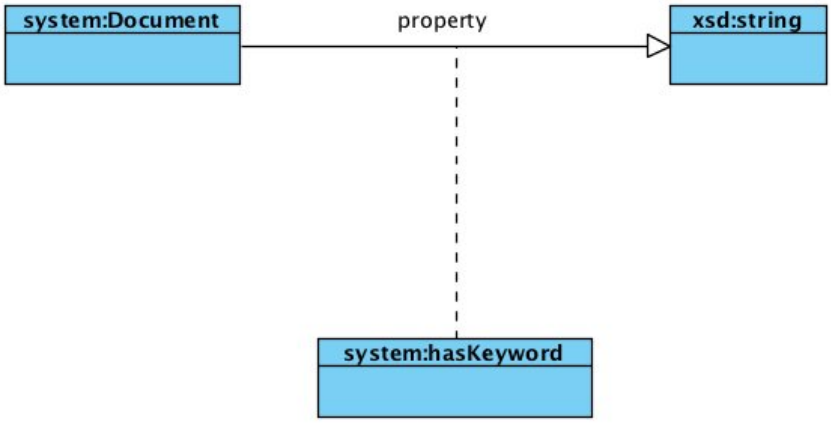


Figure 4.23. The property *system:hasKeyword* defined in the System Ontology

system:hasKeyword

```
<owl:DatatypeProperty rdf:ID="hasKeyword">
   <rdfs:label xml:lang="en">Has Keyword</rdfs:label>
   <rdfs:label xml:lang="fr">A mot clé</rdfs:label>
   <rdfs:label xml:lang="it">Contiene la parola chiave</rdfs:label>
   <rdfs:comment xml:lang="en">
            A property that links a document to a keyword contained in its content.
   </rdfs:comment>
   <rdfs:domain rdf:resource="#Document"/>
   <rdfs:range rdf:resource="xsd:string"/>
</owl:DatatypeProperty>
```

4.5 Indexing Pattern

4.5.1 Introduction

In this work we have explained several cases of indexing including the description of resources and the creation of the keys of their indexing. The description allows to express a personal point of view on the specific context of the resources because indexing is performed manually. The user follows a process of indexing, selecting first the ontology to use, in order to relate the resources with the elements contained within the ontology. The keys of indexing are created choosing concepts, individuals and relations from the selected ontology.

We are interested not only in "direct" associations like "resource about a topic", but also in descriptions requiring several steps.

We call *indexing pattern* a generalization of a case of indexing that allows to

follow a path within an ontology and to create the keys of indexing. The indexing patterns (hereafter *patterns*) are parts of the core of the system. They define a sequence of steps. At each step, the user interacts only with the necessary part of the ontology. The unnecessary parts are hidden. Selections at each step are inputs to the next step.

4.5.2 Definition of Pattern

The semantic description of a resource is an RDF graph, that we call Description Tree (discussed in section 4.2.3). All keys used for identifying a document in the index represent its semantic description and are written in a language based on RDF.

Let's consider the description: *documents Very Difficult*.

In section 4.1.4 we saw that semantic indexing is defined on possible queries the system can answer. The user thinks of the query in natural language *What are the very difficult documents?* The system, considering the user input, applies a mechanism of reformulation of the query based on the available ontologies, and creates a semantic description. In this example the description can be expressed in the form *documents that are Learning Objects with an Educational Category with a Difficulty level which is Very Difficult*, and contains the following triples:

```
_:lo rdf:type lom:LearningObject .
_:lo lom:hasEducational :_lec .
_:lec rdf:type lom:LomEducationalCategory .
_:lec lom:hasDifficulty lom:veryDifficult .
```

They correspond to the knowledge base created by the system, resulting in the description steps, where two instances _:lo and _:lec are specified:

```
_:lo
   rdf:type lom:LearningObject ;
      lom:hasLomEducational _:lec .
_:lec
   a lom:LomEducationalCategory ;
   lom:hasDifficulty lom:veryDifficult .
```

These two instances can be represented in the following RDF/XML form:

```
<lom:LearningObject rdf:ID="_:lo">
   <lom:hasLomEducational rdf:resource="_:lec"/>
</lom:LearningObject>
```

```
<lom:LomEducationalCategory rdf:ID="_:lec">
  <lom:hasDifficulty rdf:resource="lom:veryDifficult"/>
</lom:LomEducationalCategory>
```

An OWL knowledge base can be expressed also in abstract syntax, called OWL Semantics and Abstract Syntax (OWL S&AS [69]). Moreover, the Manchester OWL Syntax [49] is a syntax that has been designed for writing OWL class expressions. It was influenced by the OWL S&AS. For convenience, we give the Manchester OWL Syntax representation of the two individuals _:lo and _:lec.

```
Individual: _:lo
  Types: lom:LearningObject
  Facts lom:hasLomEducational _:lec
```

```
Individual : _:lec
  Types: lom:LomEducationalCategory
  Facts: lom:hasDifficulty lom:very_difficult
```

The representation of the individual _:lo is the particular case of an element of description well identified by all its information. It is an instance of an entry point of the LOM ontology. The *Facts lom:hasLomEducational* is specified once the type of the individual _:lo is given, *lom:LearningObject*. The same individual _:lo could be represented by a description with other *Facts*, because the LOM ontology defines other properties whose domain is *lom:LearningObject*. Thus, the representation of the individual _:lo depends on parameter <T> and can be generalized by :

```
Individual: _:lo
  Types: lom:LearningObject
  Facts <T> _:i
```

The type for the individual _:lo could be any entry point C, defined for the LOM Ontology. It is better to rename the individual _:lo by _:d (a document). Then, the generalized representation of an individual addressed by the description can be given by the following pattern:

```
Individual: _:d
  Types: <C>
  Facts <T> _:i
```

Definition 4.9. *(Description Template, $\mathcal{D}$)*

We define *Description Template*, the generalized description of a resource. Its form is expressed in terms of the Manchester Syntax, extended with parameters. Let $\mathcal{R}_i$ be the generalized description of an individual i in terms of the Manchester Syntax. The *Description Template* $\mathcal{D}$ corresponds to:

$$\mathcal{D} = \bigcup_{i=1,n} \mathcal{R}_i$$

$\mathcal{D}$ contains blank nodes and parameters. The blank nodes correspond to the individuals representing the elements of description. The values of the parameters $< T_i >$, are fixed by users during the steps followed for creating the description. At step i, the value of the parameter $< T_i >$ is given by the user's choices at step $i-1$.

The first individual defined in $\mathcal{R}_1$ has an entry point for its type. In case of a description about a resource itself, for selecting the entry point it is necessary to choose first the ontology of the domain.

The values of the parameters correspond to user's choices. For that, users need to have at their disposal available options extracted from the chosen domain ontology. Generally, the options are provided to users as results of SPARQL queries.

Definition 4.10. *(Graph Template, γ <T>)*

Let γ be a SPARQL graph pattern (see definition 2.8). A Graph Template, γ <T>, is a graph pattern containing the parameter <T>. A variable appearing in γ <T> is mapped by the query engine on a list of elements. The user'sused to determined a property choice constitutes the value of a corresponding parameter at the next step.

For example, the following graph template γ <C>:

$$\gamma \text{ <C>} = \{\text{?p rdfs:domain <C>}\}$$

contains the parameter <C> representing a concept and the variable *?p* used to determine a property.

A grounded graph template does not contain any parameters and is a graph pattern.

Definition 4.11. *(Query Template, S<T>)*

Let (γ, D, Q) be a SPARQL Query (see definition 2.9), where γ is a graph pattern, D a data set and Q a type of query.

A *Query Template* is a query where the graph pattern has been substituted by a Graph Template.

In our case, the dataset is always reduced to a single ontology $\mathcal{O}$. The query form Q is always of type SELECT. The one parameter corresponding query template is then :

$$\text{S<T>} = (\gamma \text{ <T>}, \{\mathcal{O}\}, \mathit{SELECT}\ W)$$

Definition 4.12. *(User Process, $\mathcal{P}$)*
The User Process, $\mathcal{P}$ is the sequence of steps necessary for determining the values of the variables. It is composed of a sequence of assignments involving either *Query Templates*, or other types of user inputs.

Definition 4.13. *(Algorithm, $\mathcal{A}$)* An *Algorithm* is the sequence of computations used for creating the keys of indexing. It contains some lines of pseudo-code.

The algorithm works on the elements assigned during the User Process.

Definition 4.14. *(Indexing Pattern)*
We call Indexing Pattern a triple
$\mathcal{IP} = (\mathcal{D},\mathcal{P},\mathcal{A})$ where:

- $\mathcal{D}$ is a *Description Template*;
- $\mathcal{P}$ is a *User Process*;
- $\mathcal{A}$ is an *Algorithm*.

4.5.3 Indexing Pattern on a Concept

In this section, we explain the details of the pattern addressing the indexing on a concept of an ontology. This pattern is related to the case of indexing we have called *Indexing on a concept* in section 4.3.4.2. The pattern of a knowledge base associated to the description of a resource treating of a concept is represented in the following figure (figure 4.24):

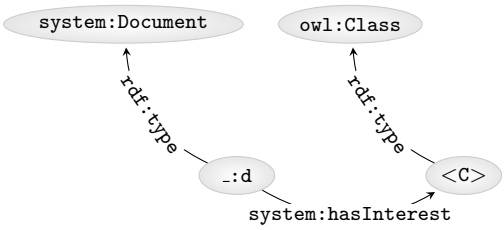


Figure 4.24. Indexing Pattern on a concept.

The user intends to describe a resource based on its content. For this case of indexing it is necessary to use the System Ontology. The resource is represented as a *system:Document* and the first property used to describe the resource is *system:hasInterest*. The elements of description are all determined, except the final concept that the user has to select within a domain ontology. The corresponding description is composed of the following triples:

_:d rdf:type *system:Document* .
_:d system:hasInterest <C> .
<C> rdf:type *owl:Class* .

Then we can define the *Description Template*, $\mathcal{D}$ as follows:

$\mathcal{D}$

```
Individual: _:d
   Types: system:Document
   Facts: system:hasInterest <C>
```

The elements of description are determined through individual *_:d*. The user selects first the ontology of the domain. In order to fix the parameter <C>, she is required to choose a concept. In fact, the parameter is determined by user's selections during the procedure defined by the following *User Process*, $\mathcal{P}$:

$\mathcal{P}$

```
O ← userOntologyChoice()
S←(γ , {O}, select ?cl)
   with γ = { ?cl rdf:type owl:Class .}
<C> ← user(res(S))
```

First, the user chooses the ontology where the concept she needs is located (method `userOntologyChoice()`). Then the system defines the graph pattern `S`. As the select clause has one variable (`?cl`), when performing the query on the dataset `{`$\mathcal{O}$`}` the results is a list of concepts (`res(S)`) where the user selects the concept she is interested in (`user()`). The process returns the class C representing the concept the user intends to use for describing the resource.

The keys of indexing determined through this pattern are generated by the Algorithm $\mathcal{A}$. The algorithm receives as an input the output of the *User Process* that is the result of user's choices. The algorithm contains the following pseudo-code:

Algorithm 4.1 Calculate Keys for the *Indexing Pattern on a concept*
Require: C
Ensure: Key
  Key = $\{rdf : type, system : Document\}$
  append to Key $\{system : hasInterest, C\}$

We do not treat explicitly the case of a property because we consider the indexing on a property in the same way as indexing on a concept.

4.5.4 Indexing Pattern on an Individual

This pattern is related to the case of indexing we called *Indexing on an individual* in section 4.3.4.4. The pattern of a knowledge base associated to the description of a resource is represented in the following figure (figure 4.25) where <i_v> represents the individual choosen by the user and <V> its type:

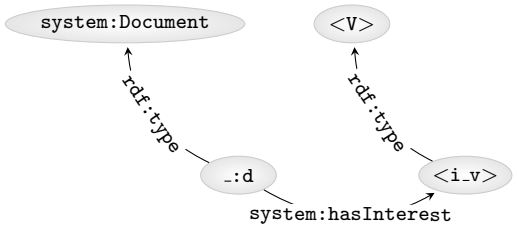


Figure 4.25. Indexing Pattern on an individual.

The corresponding description is composed of the following triples:

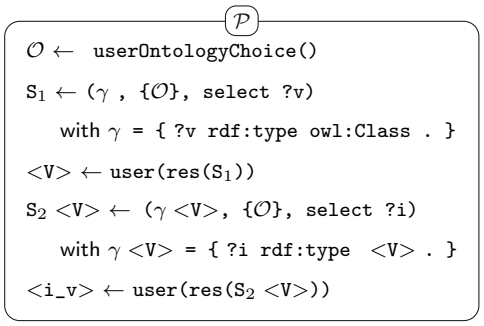
_:d rdf:type *system:Document* .
_:d system:hasInterest <i_v> .
<i_v> rdf:type <V> .

Then we can define the *Description Template*, $\mathcal{D}$ as follows:

$\mathcal{D}$

```
Individual: _:d
   Types: system:Document
   Facts: system:hasInterest <i_v>
Individual: <i_v>
   Types: <V>
```

The user selects first the ontology of the domain. Then she is required to choose a concept and in conclusion one of its individuals. The parameter <i_v> is determined by the user selections during the procedure defined by the following *User Process*, $\mathcal{P}$:



The methods `userOntologyChoice()`, `res(S)` and `user()` have the same meaning as in the previous case. The user process returns the following data:

- $< V >$, the concept representing the type of the individual to select;
- $< i\_v >$, the individual chosen by the user.

The keys of indexing determined through this pattern are generated by the Algorithm $\mathcal{A}$. The algorithm gets as input the output of the *User Process*, that is the result of user choices. The algorithm contains the following pseudo-code:

Algorithm 4.2 Calculate Keys for the *Indexing Pattern on an individual*

Require: V, i_v
Ensure: Key_initial, Key_extended
Key_initial = $\{rdf : type, system : Document\}$
Key_extended = $\{rdf : type, system : Document\}$
append to Key_initial $\{system : hasInterest, i\_v\}$
append to Key_extended $\{system : hasInterest, V\}$

The generated keys of indexing are in the following form:

Key_initial:
{rdf:type,system:Document}
{system:hasInterest, <I>}

Key_extended:
{rdf:type,system:Document}
{system:hasInterest, <C>}

4.5.5 Indexing Pattern on a Keyword

This pattern is related to the case of indexing we have called *Indexing on a keyword* in section 4.3.4.5. The knowledge base associated to the description is represented in the following figure (figure 4.26):

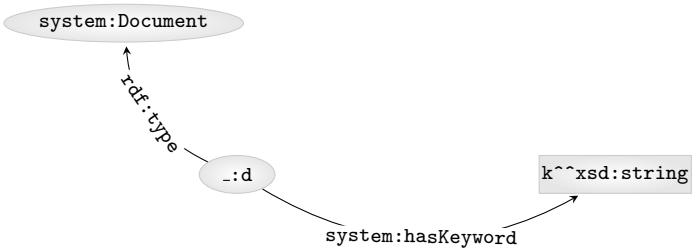


Figure 4.26. Indexing Pattern on a keyword.

The elements of description concerns the content of the resource to be indexed. The path ends on a keyword the user inserts manually.

The corresponding description is composed of the following triples:

_:d rdf:type *system:Document* .
_:d system:hasKeyword <k>^^xsd:string .

Then we can define the *Description Template*, $\mathcal{D}$ as follows:

$\mathcal{D}$

```
Individual: _:d
   Types: system:Document
   Facts: system:hasKeyword <k>
```

the following *User Process*, $\mathcal{P}$:

$\mathcal{P}$

```
<k> = user_input()
```

There is no ontology of domain involved in this pattern. The only ontology required is the System Ontology.

The keys of indexing determined through this pattern are generated by the Algorithm $\mathcal{A}$. The algorithm gets as input only the text provided by the user, inserted through the *User Process*. The algorithm contains the following pseudo-code:

Algorithm 4.3 Calculate Keys for the *Indexing Pattern on a keyword*

Require: k
Ensure: Key
Key = $\{rdf : type, system : Document\}$
append to Key $\{system : hasKeyword$,k$\}$

No extension mechanisms are applied in this case. The key has the following form:

```
Key:
  {rdf:type,system:Document}
  {system:hasKeyword, <k>}
```

4.5.6 Iterative Indexing Pattern

This pattern is related to the case of indexing that we have called *Iterative Indexing* in section 4.3.4.7. The indexing process concerns the resource itself more than its content, and is composed of a sequence of steps. First, the user chooses the ontology of the domain that, in her opinion, contains the elements of description for the resource to be indexed. Then, the user chooses the entry point in the ontology. In each of the following steps the user chooses a property whose domain is the last selected concept (the type of the resource in the first step). The system looks for the range of this property and :

- proposes the instances of this concept if any such instance occurs in the ontology,
- and proposes to continue.

If the user continues the process, the system creates a virtual individual of the range of the last selected property else the process stops because the user has chosen a real individual of the ontology. The user can also give up the process.

Let's consider as an example the process completed in 2 steps. The pattern of a corresponding resource description is represented in the following figure (figure 4.27):

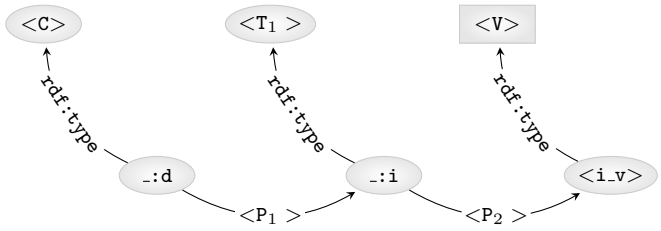


Figure 4.27. Iterative Indexing Pattern with 2 steps.

The pattern is called *with 2 steps* because it involves the selection of two properties: P_1 and P_2.

The pattern involves two virtual individuals, *_:d* and *_:i* that the system has to create and the final choice of the user <i_v>. The type of the first virtual individual is an entry point of the ontology of the domain chosen by the user. The second one is created as an instance of the concept $< T_1 >$. The corresponding description is composed of the following triples:

```
_:d rdf:type <C> .
_:d <P1 > :_i .
_:i rdf:type <T1 > .
_:i <P2 > <i_v> .
```

We can define the *Description Template*, $\mathcal{D}$ as follows:

$\mathcal{D}$

```
Individual: _:d
   Types: <C>
   Facts: <P1 >
Individual: _:i
   Types: <T1 >
   Facts: <P2 > <i_v>
```

The parameters are determined by the user selections during the procedure defined by the following *User Process*, $\mathcal{P}$:

$\mathcal{P}$

```
O ← userOntologyChoice()
<C> ← user(entry_point(O))
S1<C> ← (γ<C>, {O}, select ?p ?r)
  with  γ<C> = {
    ?p rdf:type owl:ObjectProperty .
    ?p rdfs:domain <C> .
    ?p rdfs:range ?r . }
<P1, T1>← user(res(S1<C>))
S2<T1 >← (γ<T1>, {O}, select ?p ?v)
   with γ<T1> = {
    ?p rdf:type owl:ObjectProperty .
    ?p rdfs:domain <T1 > .
    ?p rdfs:range ?v . }
<P2, V> ←  user(res(S2<T1>))
S3<V> ← (γ<V>, {O}, select ?i)
   with γ<V> = {?i rdf:type <V> .}
<i_v> ← user(res(S3<V>))
```

The line *user(...)* expresses that user input is required. The first and second lines concern the selection of the ontology of the domain and one of its entry points. It is important to observe that in this *User Process* the two *Query Template* $\mathtt{S}_1$ and $\mathtt{S}_2$ have the *Graph Template* $\gamma <\ldots>$ of the same form and consist in querying properties with a specific domain and their ranges. The two steps followed by a user for describing resources through this pattern are guided by the queries $\mathtt{S}_1$ and $\mathtt{S}_2$ that extract from the ontology the useful elements at each step. The user must choose the elements needed for the description by selecting among the results provided by the queries. The final query $\mathtt{S}_3$ determines the end of the user process because the user selects a real individual defined in the ontology.

The iterative pattern with 2 steps is a particular case of the general *Iterative Pattern*. The general case is structured in n steps because the user can follow an indefinite number of steps before choosing the final individual that ends the description. The description is composed of an indefinite number of triples and can be represented as follows:

_:d rdf:type <T_0 > .
_:d < P_1 > :_i_1.
_:i_1 rdf:type <T_1 > .
_:i_1 < P_2 > :_i_2.
...
_:i_{n-1} rdf:type <T_{n-1} > .
_:i_{n-1} < P_n > <i_v> .

The knowledge base associated with the description is represented in the following figure (4.28):

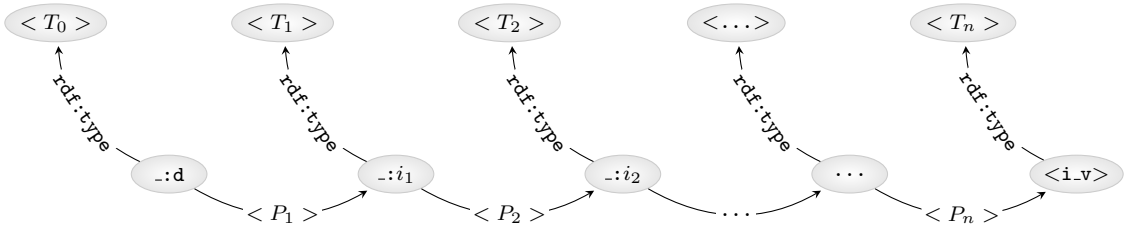


Figure 4.28. Iterative Indexing Pattern with n steps.

The *Description Template*, $\mathcal{D}$ is defined as follows:

$\mathcal{D}$

```
   Individual: _:d
      Types: <T0>
loop (k=1,n)
      Facts: <Pk> _:ik
   Individual: _:ik
      Types: <Tk>
end loop
    <i_v> ← _:in
    <V> ← <Tn>
```

The type of the first individual *_:d* is an entry point of the domain ontology chosen by the user. Then, *n-1* virtual individuals ($\_:i_k$) must be created, connected by a property.

The *User Process*, $\mathcal{P}$ is the following:

$\mathcal{P}$

```
O ← userOntologyChoice()
T0 ← user(entry_point(O))
k ← 0
i_v ← null
repeat
     k ← k++
    Sk<Tk−1>←(γ<Tk−1>,{O}, select ?p ?r)
       with γ<Tk−1>= {
        ?p rdf:type owl:ObjectProperty .
        ?p rdfs:domain <Tk−1> .
        ?p rdfs:range ?r . }
    <pk, Tk>← user(res(Sk<Tk−1>))
    SF<Tk>←(γ<Tk>, {O}, select ?i)
       with γ<Tk>= { ?i rdf:type <Tk> . }
    if (res(SF<Tk>))≠ ∅
      <i_v> = user(res(SF<Tk>))
until <i_v> ≠null
```

The statement *repeat ... until* repeats the queries $\mathtt{S}_k$ $<\mathtt{T}_{k-1}>$ and $\mathtt{S}_F$ $<\mathtt{T}_k>$. The first allows one to extract from the ontology the useful elements that the user must select to continue the steps. The last query allows to determine the final individual. The value chosen among the results of this query is used as an exit condition for the statement *repeat ... until*. The user process returns the following data:

- T_0, the entry point of the domain ontology chosen by the user;
- p_k (k = 1,n), the properties chosen by the user through the n steps of indexing;
- $< i\_v >$, the individual chosen by the user at the end of the process of indexing.

The keys of indexing determined through this pattern are generated by the Algorithm $\mathcal{A}$. The algorithm gets as input the output of the *User Process*, that is the result of user choices. The algorithm contains the following pseudo-code:

Algorithm 4.4 Calculate Keys for the *Iterative Indexing Pattern*

Require: T_0, n, p_k, i_v
Ensure: Key_initial, Key_extended
  Key_initial = {$rdf : type$,T_0}
  Key_extended = {$rdf : type$,T_0}
  for all i=1,(n-1) **do**
    append to Key_initial {p_i}
    append to Key_extended {p_i}
  end for
  append to Key_initial {p_n,i_v}
  append to Key_extended {p_n}

that generates the keys in the following form:

Key_initial:
  {rdf:type, <T_0 >}
  {<p_1>}
  {...}
  {<p_n>, <i_v>}

Key_extended:
  {rdf:type, <T_0 >}
  {<p_1>}
  {...}
  {<p_n>}

4.5.7 Iterative Indexing Pattern Involving a Virtual Individual

This pattern is related to the case of indexing that we called *Iterative Indexing involving a virtual individual* in section 4.3.4.8. This pattern is similar to the *Iterative Pattern*. The difference is in the last step of the description. This pattern ends when the user defines a virtual individual and selects one of its properties.

The description related to this pattern is composed of an indefinite number of triples and can be represented as follows:

_:d rdf:type <T_0 > .
_:d < P_1 > :_i_1.
_:i_1 rdf:type <T_1 > .
_:i_1 < P_2 > :_i_2.
...
_:i_{n-1} rdf:type <T_{n-1} > .
_:i_{n-1} < P_n > _:i_n .
_:i_n rdf:type <T_n > .
_:i_n < P_f > <k>^^xsd:string .

The knowledge base associated to the description is represented in the following figure (4.29):

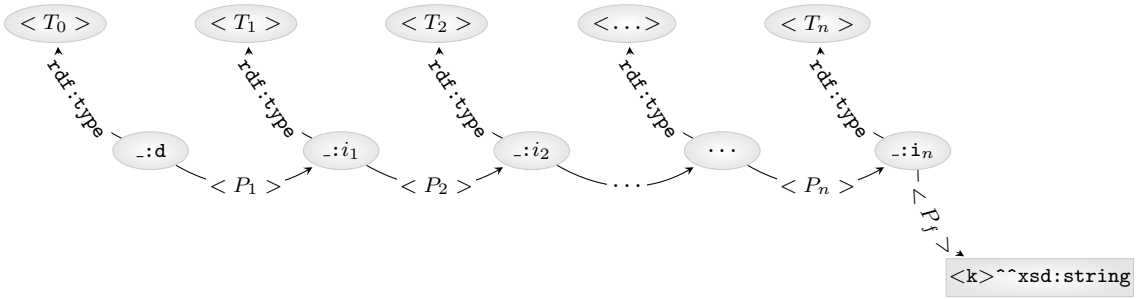


Figure 4.29. Iterative Indexing Pattern with n steps involving a virtual individual.

The *Description Template*, $\mathcal{D}$ is defined as follows:

$\mathcal{D}$

```
   Individual: _:d
      Types: <T0>
loop (j=1,n)
      Facts: <Pj> _:ij
   Individual: _:ij
      Types: <Tj>
end loop
   Individual: _in
       Types: <Tn>
       Facts: <Pf> <k>^^xsd:string
```

The *User Process*, $\mathcal{P}$ is the following:

$\mathcal{P}$

```
O ← userOntologyChoice()
T0 ← user(entry_point(O))
j ← 0
i_v ← null
repeat
     j ← j++
    Sk<Tj−1>←(γ<Tj−1>, {O}, select ?p ?r)
      with γ<Tj−1> = {
       ?p rdf:type owl:ObjectProperty .
       ?p rdfs:domain <Tj−1> .
       ?p rdfs:range ?r . }
    <pj, Tj> = user(res(Sj<Tj−1>))
    SF<Tj>←(γ<Tj>, {O}, select ?p_f)
      with γ<Tj> = {
       ?p_f rdf:type owl:DatatypeProperty .
       ?p_f rdfs:domain <Tj> .
       ?p_f rdfs:range xsd:string .}
    <pf> = user(res(SF<Tj>))
until <pf>≠null
<k> = user_input()
<Tn>=<Tj>
```

The user process returns:

- T_0, the entry point of the domain ontology chosen by the user;
- p_j (j = 1,n), the properties chosen by the user through the n steps of indexing;
- p_f, the property for the final virtual individual;
- $< k >$, the keyword associated to the final virtual individual through the property p_f.

The keys of indexing determined through this pattern are generated by the Algorithm $\mathcal{A}$. The algorithm gets as an input the output of the *User Process* that is the result of user's choices. The algorithm contains the following pseudo-code:

Algorithm 4.5 Calculate Keys for the *Iterative Indexing Pattern involving a virtual individual*

Require: T_0, n, p_f, <k>,
Ensure: Key_initial, Key_extended_1, Key_extended_2
  Key_initial = $\{rdf : type, T_0\}$
  Key_extended_1 = $\{rdf : type, T_0\}$
  for all j=1,(n-1) **do**
    append to Key_initial $\{p_j\}$
    append to Key_extended_1 $\{p_j\}$
  end for
  append to Key_initial $\{p_n, < T_n >\}$
  append to Key_initial $\{p_f, < k >\}$
  append to Key_extended_1 $\{p_n\}$
  Key_extended_2 = $\{rdf : type, system : Document\}$
  append to Key_extended_2 $\{system : hasKeyword$,¡k¿$\}$

that generates keys of the following form:

Key_initial:
{rdf:type, <C>}
{<R_1>}
{...}
{<R_n>, <T>}
{<P>, <k>}

Key_extended_1:
{rdf:type, <C>}
{<R_1>}
{...}
{<R_n>}

Key_extended_2:
{rdf:type, system:Document}
{system:hasKeyword, <k>}

4.6 Main notions about Community

4.6.1 Introduction

Community of users interested in the same issues are frequently involved in collaborative activities of sharing and searching resources of similar types. The core aspect is that the use of heavy tools and the dependency on centralized repositories could restrain their participation. To attain such kind of collaboration, users of a community have to describe their material semantically in relation to one or more ontologies.

In the field of education, teachers use pedagogical material for supporting the progress of courses and other activities that pertain to teaching. It is important for both teachers and students to be able to easily find and bring back interesting documents. Teachers agree to share resources with other teachers or students thanks to a simple mechanism.

In this context we propose a system that allows the semantic indexing of resources and their sharing among the members of the community. It consists of a P2P infrastructure and a Web platform equipped with some tools. The documents are associated with a semantic descriptions, and then are published in the P2P network.

The life of a community is not limited to indexing, publication and exchange of resources such as documents. Other issues may concern the interests of the members, the tools they use and the information exchanged among the members.

4.6.2 Community Resources

It is necessary to distinguish clearly between the categories of resources involved in the system.

Definition 4.15. *(Community resources)*

We call *Community resources* the set of all kind of resources relating the life of the community. It is the union of the subsets *Documents* and *Core Resources*.

Definition 4.16. *(Documents)*

Documents is the set of the resources shared by users through the Shared Memory or stored in Personal Memory. Both memory indexes store entries containing pairs (key, URL). In the shared memory, documents are accessed through their URL.

Definition 4.17. *(Core Resources)*

Core Resources is the set of the core elements of the System, consisting of the ontologies used to create the keys of indexing, the elements bound to the community such as Personal Notes and the Wiki of the Community where users can insert and update information regarding the community in a collaborative way. People can

describe special events, add notes, etc. They must be accessible to all members. The core elements of the System require their content be inserted into the distributed index. In the Personal Memory the resources are stored in a local archive within the peer.

We decided to deal in the same way with all *Community resources*, using the same mechanism of indexing, publication and retrieval.

4.6.2.1 Documents

In our System, *Documents* are resources provided by users. For instance, the files used in teaching course are the documents related to the same course. They are contained in the Personal Memory of the owner or may be stored in the Shared Memory if the user wishes to share them with the other members of the community.

4.6.2.2 Ontologies

Ontologies are used for creating the keys of indexing. We can find details in section 4.4.

Ontologies are published in the network. For publishing an ontology, a key of indexing and a small additional description are required. The key of indexing is not decided by users but it is assigned automatically by the System. The description is the additional information integrated in the value of the index entry when the ontology is published. It includes the application domain of the ontology, the set of Entry Points, and the URI identifying the namespace name of the ontology.

Users have to retrieve ontologies when they start the system. They also have to find a domain ontology that suits their needs. The System Ontology is used for describing the *Core Resources*. The System Ontology is automatically published in the system when the system bootstraps. The System Ontology is extended with the definition of all resources included in the system.

A domain ontology is published with the key:

```
Key:
  {rdf:type, system:Ontology}
```

Its entire content is inserted in the DHT when the publication involves the Shared Memory. As with other resources, the ontologies indexed in the Personal Memory are stored in the file system of the user peer who owns the ontology. The index entry stores the file name of the ontology.

4.6.2.3 Notes

A *Note* is a free text provided by a user to include additional information in the System. The content of the Note may be any topic of interest for the user. The use

of Notes is considered of general purpose because they can carry various information, such as messages for other users, memos, comments on certain resources, etc.

An Note is published with the key:

Key:
{rdf:type, system:Note}
{system:hasKeyword, <k>}

For allowing the users to distinguish among a wide amount of Notes, we choose to allow to add keywords for indexing for a Note. The case of *Indexing on a keyword* is the base case we use for indexing a Note. The key of indexing concerning a Note contains the specific kind of document ({rdf:type, system:Note}).

The content of the Note is inserted in the DHT when the publication involves the Shared Memory. The content of a note indexed in the Personal Memory is saved within a file, stored within the file system of the user peer. The index entry stores the file name of the note.

4.6.2.4 Wiki

The community is equipped with a unique space shared by all users. This role can be identified in Wikis. A Wiki is a web site for creating and editing any number of interlinked web pages via a web browser using a simplified language of markup via an integrated text editor. In our System, the Wiki of the Community (hereafter *Wiki*) is composed of only one physical document containing several pages that may link to other resources, distributed in the P2P network.

A specific key gives access to the wiki. It is published with the key:

Key:
{rdf:type, system:Wiki}

The Wiki is a document of the System. It is composed of a *template* file containing the skeleton of the Wiki with only the essential structure, without any content. At the beginning, when a new community is created, the System starting from the *template*, publishes the Wiki in the Shared Memory. The users of the community may start to work on the shared Wiki, retrieving it from the network through the key of research. Any modification on the Wiki are saved in the same file. The modified Wiki may be again published in the Shared Memory thanks to the embedded functionalities.

In the Wiki, we have substituted the usual static reference links with semantic links (described in section 4.6.4.3), representing the resources distributed in the P2P network. Such resources are semantically indexed, through keys of indexing, and are stored in the distributed index and then can be retrieved by the wiki thanks to analogous requests.

We decided not to index the Wiki in the Personal Memory memory because the wiki is a unique resource shared by all users an there is no need to keep other local copies.

4.6.3 Semantic Desktop

We consider users of a community who share their own resources that provide certain semantic keys of indexing. They wish to retrieve the same resources or other resources through keys of research created in the same way. They do not want to deal with complicated tools or depend on centralized repositories. The resources and their semantic indexing keys are published in the P2P network. Otherwise they are stored in the Personal Memory if users decide to keep them for a private use. Publication of semantically described document in P2P networks was presented as a real challenge in [30]. Students and other teachers can discover the resources by making queries based on the descriptions created through the use of ontologies. In this scenario, the choice of the ontologies is fundamental and ontologies must be shared with the community.

Our system allows to manage resources of various types, not necessarily textual. Semantic annotations represent objective information about resources (nature, concepts of scientific domain, etc.) but can also represent a point of view on documents (difficulty level, usefulness in some context, etc.). Centralized memories may have some disadvantages when they need to be filled up. A distributed system like this could be a solution to this issue.

The resources of our system are considered partly like in the Gnowsis System [88] where each resource is identified by a Uniform Resource Identifier (URI) and all data is accessible and queryable as an RDF graph. In order to create a network of users in the sense of community, in our opinion is necessary to create a platform as a semantic desktop equipped with a set of tools at users' disposal. Even Tim Berners Lee did not really envision the World Wide Web as a hypertext delivery tool, but as a tool to make people collaborate [31]. The users entry point to the system is a Semantic Desktop (see figure 4.30), a Web-like shaped user interface resembling the desktop of a traditional operating system.

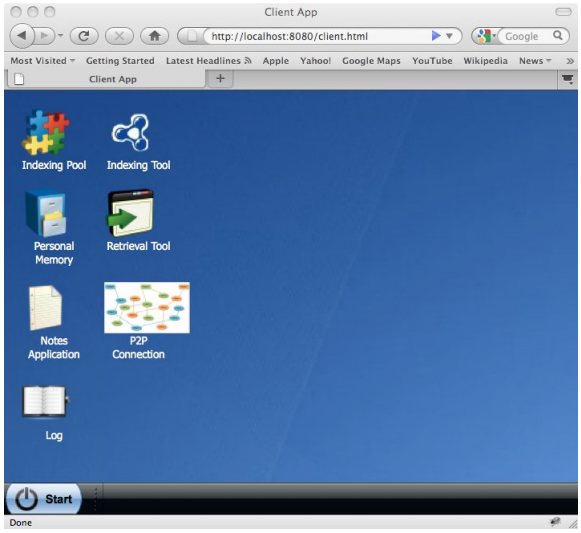


Figure 4.30. The Semantic Desktop

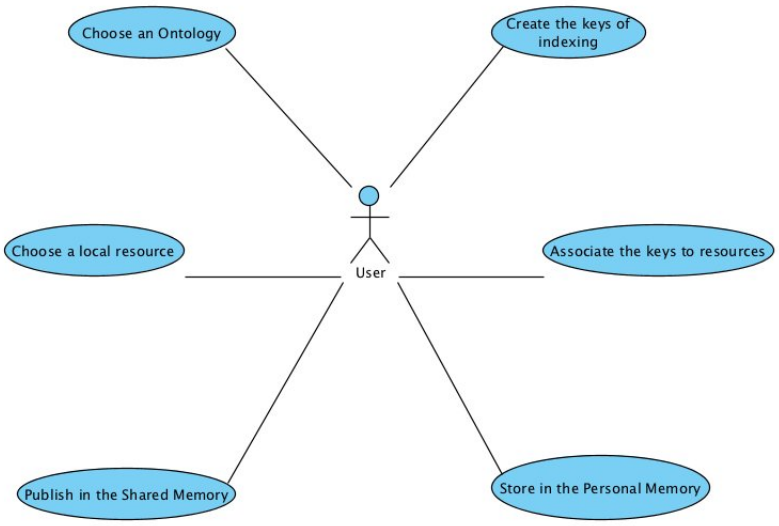


Figure 4.31. Use cases

The figure 4.31 shows the actions allowed to users within the Semantic Desktop. The actions concern the description of resources for their publication in the Shared Memory and in the Personal Memory. For efficiently accessing the shared and personal resources, ontologies used for indexing are shared by all members of the community. Descriptions of local resources are created (*Create a description*), and become useful when published (*Choose a local resource*) into the shared memory. When the system starts the ontologies are discovered automatically. A user gets an

updated list of available ontologies and can select the ones needed (*Select an ontology*). This tool also allows for an intuitive navigation inside ontologies and prepares a semantic description of a resource in the background. Such descriptions must also be used to retrieve relevant resources from both memories. The association between a resource name and a description (*Associate description to resource*) launches the indexing of the resource in the local memory (*Store in personal memory*) and/or the publication in the shared memory (*Share the classified resource*).

4.6.4 Semantic Links

4.6.4.1 Traditional Web Pages

A Web page [3] is a resource of information that is suitable for the World Wide Web and can be accessed through a web browser. This information is usually in HTML or XHTML format, and may provide navigation to other web pages via hypertext links (or hyper-links). An hyper-link is a reference or navigation element in a document to another section of the same document or to another document that may be on or part of a (different) domain. Web pages may be retrieved from the local computer or from the remote web server. The web server may restrict access only to a private network, e.g. a corporate intranet, or it may publish pages on the World Wide Web. Web pages are requested and served from web servers using Hypertext Transfer Protocol (HTTP). Web pages may consist of files of static text stored within the web server's file system (static web pages), or the web server may construct the (X)HTML for each web page when it is requested by a browser (dynamic web pages). Client-side scripting can make web pages more responsive to user input once in the client browser.

Hypertext most often refers to text on a computer that will lead the user to other, related information on demand. Hypertext overcomes some of the limitations of written text. Rather than remaining static like traditional text, hypertext makes possible a dynamic organization of information through links and connections (called hyperlinks). Hypertext can be designed to perform various tasks; for instance when a user "clicks" on it or "hovers" over it, a bubble with a word definition may appear, a web page on a related subject may load, a video clip may run, or an application may open.

4.6.4.2 Typed Links

"*The Semantic Web isn't just about putting data on the web. It is about making links, so that a person or machine can explore the web of data. With linked data, when you have some of it, you can find other, related, data*" (Tim Berners-Lee)[11].

[3] http://en.wikipedia.org/wiki/Web_page

In Web pages a *typed link* is a link to another resource including more information about the link. It is not only an indication of the existence of a resource, but a typed link may also specify that the resource includes some characteristics such as, for instance, be the substitute version for the resource in which the link occurs. User agents, search engines, etc. may interpret the typed links in a variety of ways. For instance, a user might acts in certain ways, such as searching only specific types of links. It may also allow browsers or browsing software to pre-fetch linked resource according to user needs.

Html 4 [29] supports typed links providing the *rel* (forward link) and *rev* (reverse link) attributes in $< link >$ and $< a >$ tags. For instance, the line

$$< \text{link rel="index" href="link\_A"} >$$

means that the *link_A* refers to a resource providing an *index* to the current resource;

or the line

$$< \text{link rel="section" href="link\_B"} >$$

means that the *link_B* refers to a document that serves as a *section* in a collection of documents. The complete list of values for these attributes may be found in [29].

The *rel* and *rev* attributes play complementary roles: the *rel* attribute specifies a forward relationship and the *rev* attribute specifies a reverse relationship. Considering two documents A and B, the following meanings are equivalent:

$$\text{Document A: <link rel="}\ldots\text{" href="doc\_B" >}$$

$$\text{Document B: <link rev="}\ldots\text{" href="doc\_A" >}$$

Browsers do not use these attributes but search engines may benefit from typed links in getting more information about a link.

4.6.4.3 Distributed Semantic Links

Definition 4.18. *(Semantic Link)*

We call *Semantic Link* the link contained in a Web page that links to other resources that may be a Web page or other resources such as documents, videos, etc. The link is semantically described and a *key* is created for its indexing.

For instance, the line

$$\text{<link onClick="computeSemanticLink(key)" >}$$

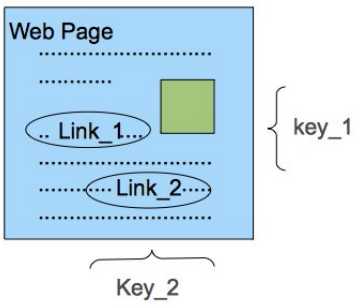


Figure 4.32. Distributed Links

refers to a *Semantic Link* identified through the key of indexing *key*. For determining the resource addressed by the key, the JavaScript function *computeSemanticLink(key)* is evoked.

```
<script type="text/javascript">
      function computeSemanticLink(key) {
         ...
      }
</script>
```

The *onclick* attribute of the HTML <link> tag allows to capture the event that occurs when the pointing device button is clicked over the link in the Web page.

In our system we consider web pages as shared resources. Web pages are reachable through their URLs. We choose to publish them in the DHT by a key of indexing. A web page can link other pages, through hyper-links, that are also considered shared resources identified by a key of indexing and an URL.

Definition 4.19. *(Distributed Semantic Link)*
We call *Distributed Semantic Link* the link contained in a Web page that links to another resource. The link is semantically described and a *key* of indexing is created for its publication in the DHT.

To retrieve linked resources, the system uses the key of indexing for making queries.

In figure 4.33 a *Web Page* is a resource published in the system through some keys. The *Web Page* is physically stored within the *Peer s*. The *Web Page* refer to two resources via *Link 1* and *Link 2*. The *Link 1* is published under the *Key 1* and is stored within the *Peer x*, whereas the *Link 2* is addressed by the *Key 2* and is stored within the *Peer y*. These keys are used by the system do discover the resources related to the two links.

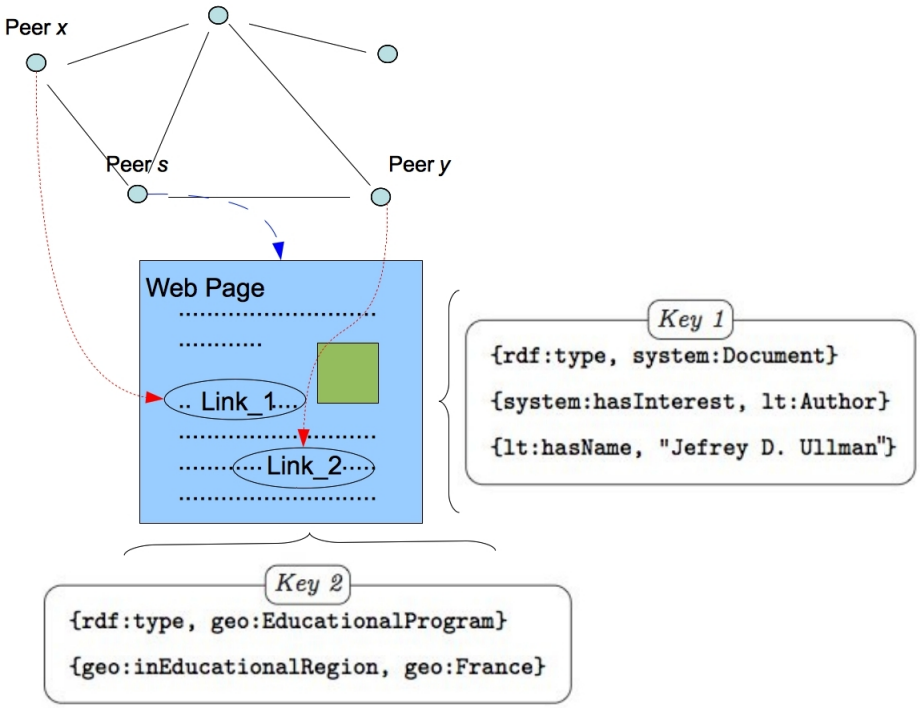


Figure 4.33. Distributed Semantic Links

The advantages of such application is that resources are always scattered on the network. Of course this is one of the benefits of the P2P paradigm. Moreover, as the same key of indexing may address more than one resource, a link could address more resources. The number of addressed resources is dynamic because the activities of indexing and publishing made by users of the community make the *Shared Memory* grow.

Chapter 5

Implementation

This chapter describes the architecture and the implementation of the System. First, an overview of the System is given. Then the design of the proposed architecture and its three component layers, *Front-end*, *Services* and *Function*, are introduced. The development is described in detail in order to explain the choices for the component parts. The chapter ends with a discussion of some ideas for future development.

5.1 System Overview

In this chapter we describe the architecture and the development of the System that we designed for simplifying the indexing and the sharing of resources. The system is depicted in figure 5.1. The activities of a *P2P* network take place on the *Internet* (shown in the middle of the figure). People are grouped into communities in which members are connected to each other by the same motivation to communicate, collaborate and share their interests. Users are interested in maintaining beneficial contacts with other members and in sharing real and useful resources that concern the whole community.

5.1.1 Community

The *community* (outlined in figure 5.2) is the point of aggregation of *users* who share interests and/or activities. In our case, this community is intended as a distributed network of users. The resources shared within the community are contained within a *shared memory*. In the context of the community, the meaning of the *memory* is abstract. It represents the wide archive of knowledge that is owned by the community members. Such knowledge can be either private or shared by all users. The memory is the space where everybody can store resources. Users have to query an *index* to find the resources in that interest them.

Figure 5.1. System Overview

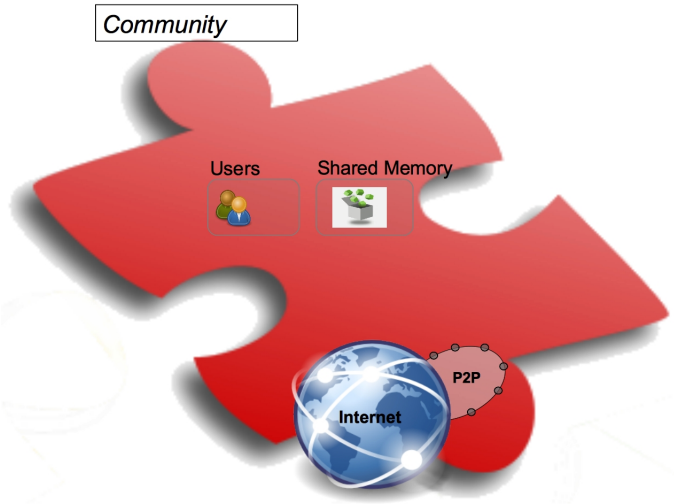


Figure 5.2. The Community

Each user takes part of the community through a *user peer* that constitutes the user entry point in the system. The user peer provides a User Interface that can be accessed from anywhere in the Internet. It ensures users the necessary facilities for participating in the community.

5.1.2 User Peer

The *user peer* is the entry point in the system. The user accesses the system and interacts with it by means of tools included in the graphical user interface (designed as a Web application). The user peer is shown in figures 5.3 and 5.4. A user may interact with the system for:

- indexing resources;
- sharing resources;
- searching resources.

The tools are intended mostly for resource management and interaction with the private and the shared memory. The peer physically stores the resources in the private and shared memories. The shared resources are stored in a public area and are available via a local web server, as it is common with resources accessed on the Web.

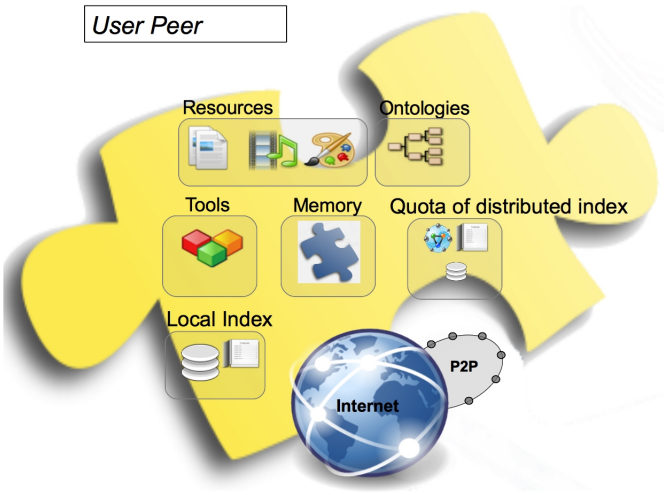


Figure 5.3. The user peer

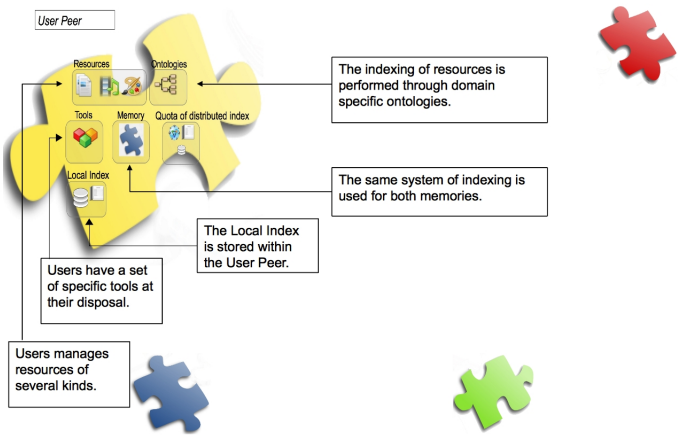


Figure 5.4. Features of the user peer

Each user peer owns a *memory* that contains a private part for archiving personal resources. The *memory* also contains a public part that stores those documents that are shared with all members of the community. The *memory* (see figure 5.5) is divided in two parts: the *Personal Memory* and the *Shared Memory*. The former stores resources that a user keeps in the private archive. Depending on the user's intention, some of these resources can also be published in the *Shared Memory*. The *Shared Memory* contains the resources that each user has published in the network. The abstract concept of *Community Memory* corresponds to the union of all shared

memories of all the peers. Even if we distinguish between local and distributed indices, both memories are managed by the same mechanism of indexing. In the *Shared Memory*, a resource is accessed either through its URL or through its entire data content. The resources contained in the *Personal Memory* are stored in a local archive within the peer. In this memory, resources may be files or Notes. Resources stored as files are accessed through their reference in the local file system.

The shared memory is the part of the system that is accessible by all users. The users can discover and retrieve resources that interest them.

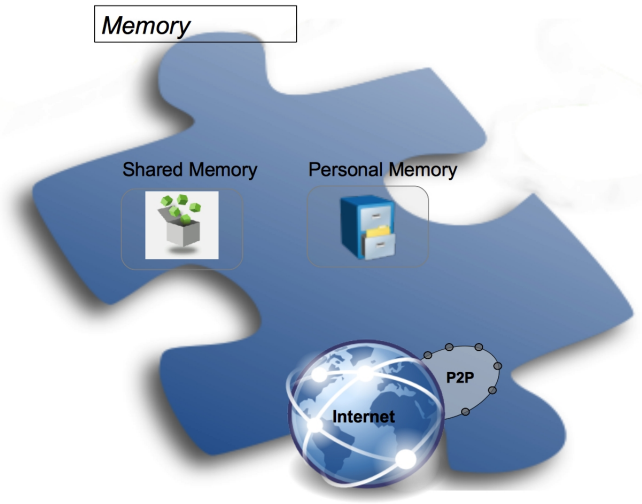


Figure 5.5. The Memory

The user peer, as part of a P2P community, contains a quota of the whole distributed index. Performance of the P2P infrastructure guarantees the availability of the distributed index. Resource parts of the private memory are indexed with another index stored locally.

The *index* is the part of the community that contains the information that semantically describes the resources (the keys of indexing) and their physical links. Given a key of indexing, the information necessary for accessing associated resources can be retrieved from the index. Unique mechanisms of indexing are used for local resources and for the resources shared by the community. Therefore, in the system we distinguish between a *Local Index* and a *Distributed Index*. The *Local Index* refers the resources kept private by users. The *Distributed Index* concerns shared resources and is distributed via a data structure called DHT. In figure 5.6 the indexes are part of each peer who belong to the community.

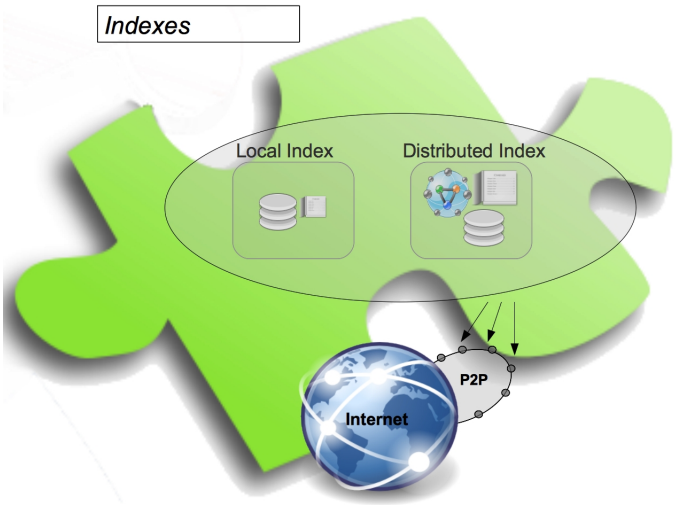


Figure 5.6. The indexes

Relationships between *Memory*, *Index* and *Community* are shown in figure 5.7.

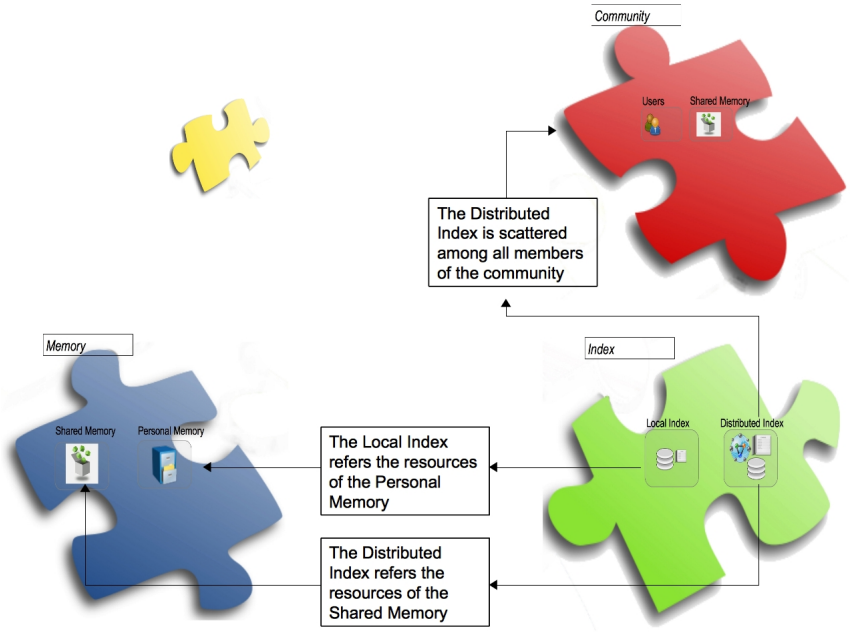


Figure 5.7. Relations among Memory, Index and Community

See figure 5.8 for connections between *Memory*, *User Peer* and *Index*.

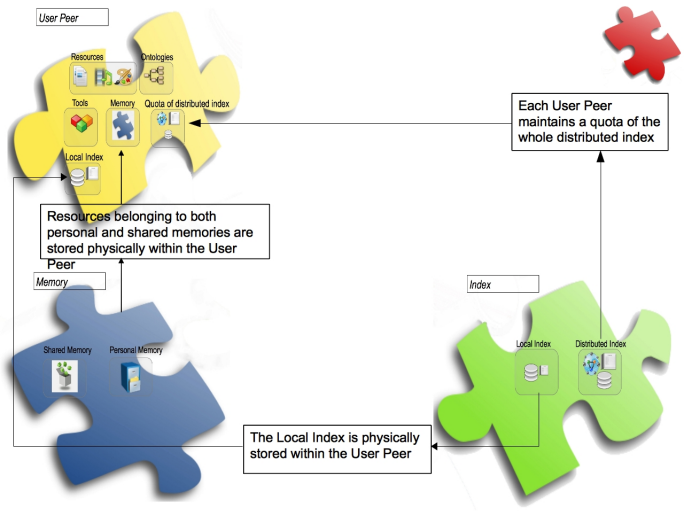


Figure 5.8. Relations among User, Memory and Index

All users may participate in the community by using a user peer.

5.1.3 Joining the Community

A new user who is interested in the community, has to join the network and become a member of the network of peers (see figure 5.9).

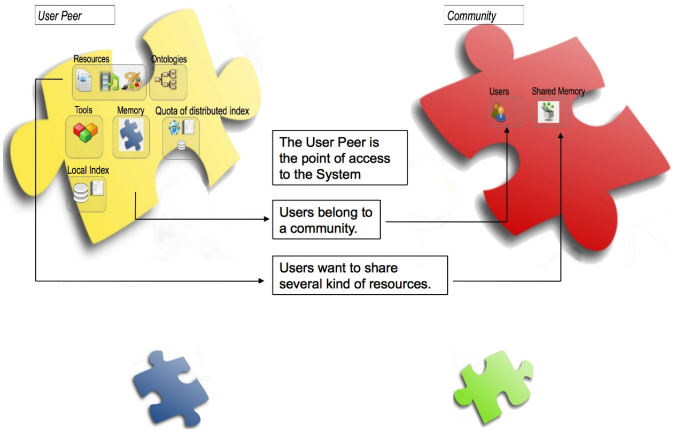


Figure 5.9. Relations between User Peer and Community

There are several ways to join a P2P network and many aspects have to be considered:

- a peer needs to know at least one neighbour (current member); each peer owns a personal list of neighbours;
- the list of neighbours or most reliable peers may dynamically change as the network changes; a broadcasting message for the discovery of most reliable peers may be sent through the network;
- web cache: there is at least one reliable on-line repository of IP addresses of alive peers.

In our system, a new peer reads a list of likely candidate neighbours. The list contains references to other peers in the network who may be alive, and make it possible for the new peer to join the network.

5.2 Architecture

The user peer is a system that is composed of three layers, called *Front-end*, *Services* and *Function* (see figure 5.10). Its design follows the logical separation of presentation, application processing and data management.

The low level layer (*Function*) is composed of three modules: the *Ontology* module, the *P2P* module and the *Memory* module.

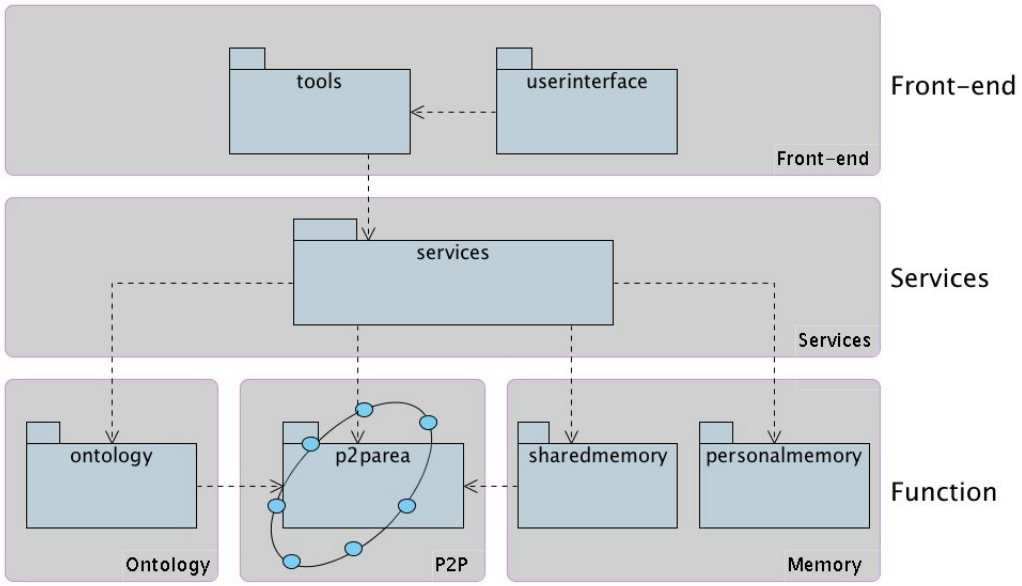


Figure 5.10. The System Architecture

The *Ontology* module is concerned with the management of ontologies and indexing patterns. The *P2P* module provides functionalities for accessing the network of peers. It constitutes the infrastructure of the system in which the community of users is scattered in a P2P network. The P2P network is composed of atomic elements called *peers* (or *nodes*). The *P2P Area* is an atomic element of the network that allows both publishing and retrieving actions on the distributed index. On the network side, the chosen technology is built on Pastry, a generic, scalable and efficient substrate for P2P applications. We use FreePastry[1] [82], the open-source implementation of Pastry. Its features allow for adapting the network to the specific needs. The *Memory* module is concerned with indexing and storing of resources within personal and shared memories. It focuses on physically storing and retrieving the resources within the two memories: the Personal Memory for local repository of resources and the Shared Memory for resources shared by the users in the network.

[1] http://www.freepastry.org/

The *Services* layer is composed of an extensible set of web services. The set of web services provides an integration substrate, giving access to the features of the nodes of the *P2P* module and the features of the *Memory* module and the *Ontology* module. The capabilities of the system are based on the SOA (Service Oriented Architecture) design principles [2] and each capacity is implemented as a web service, exploiting the features of the REST [35] paradigm. Such layers are designed to interconnect the *Front-end* layer and the low level part of the system. The system is extensible, so that it is easy to implement new capabilities by providing new web services.

The *Front-end* layer provides the user interface that allows users to interact with the system. The global concept of the *Front-end* is a Web interface that guarantees an access to the system anywhere and any time. The Web interface is designed to resemble a desktop of a traditional operating system. It is equipped with a set of *Tools* for implementing all features of the user interface. It allows a user to choose ontologies, select resources, create keys of indexing, publish and search resources, etc. It is extensible and allows improving the desktop with new applications for different purposes. The user interface is developed through the web technologies HTML, JavaScript, CSS, and is animated with AJAX techniques. We have based all these features on the ExtJs[2] JavaScript library. Each node of the community is associated with one Web interface. The user can access the system through such a Web interface. There are no constraints on connecting the system with a specific one. The system can be accessed from the Web interface of any peer. We have designed a Web application in order to provide a cross-platform solution with no requirements of specific installations. Such interface is designed as a set of views on tools and looks like a Semantic Desktop.

[2]http://www.sencha.com/products/js/

5.3 Function Layer

The *Function* layer is composed of the modules *Ontology*, *P2P* and *Memory*. Each of these modules are represented, in the next pictures, with UML packages.

5.3.1 Ontology

The *Ontology* module concerns two aspects: i) the management of the ontologies; ii) the management of the indexing patterns. These two aspects are strictly related because the indexing patterns are based on the ontologies available in the System. In figure 5.11, the *ontology* package contains the class diagram relative to the implementation of the *Ontology* module.

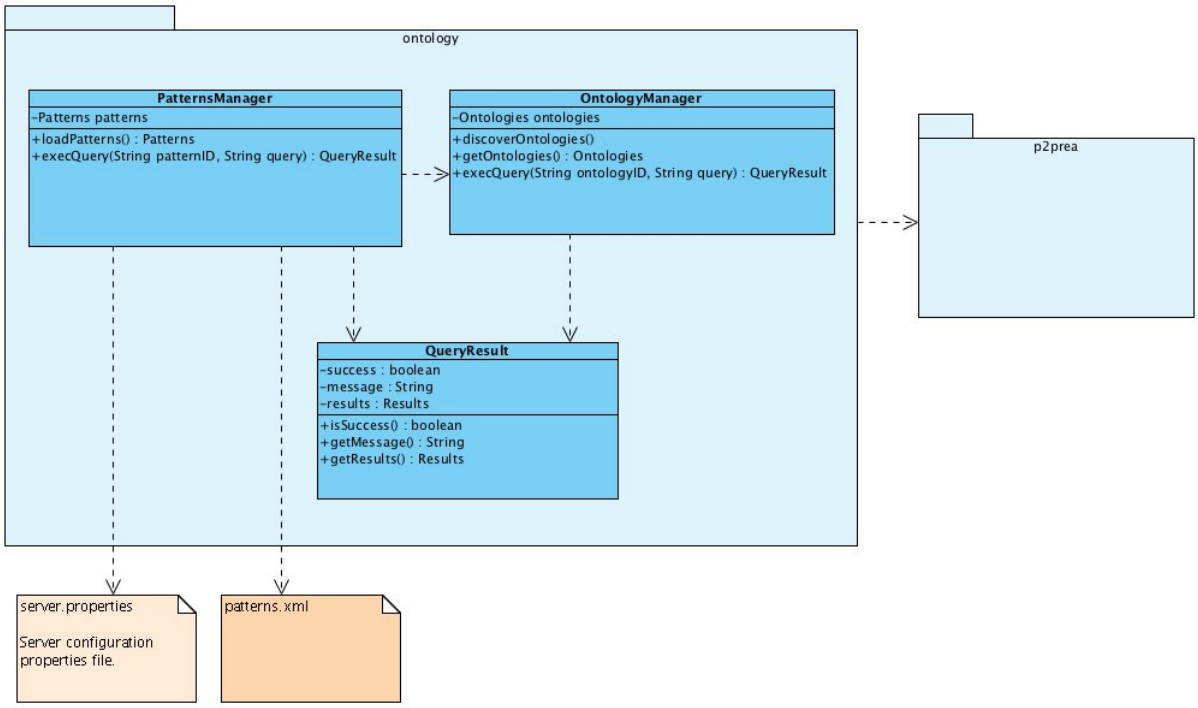


Figure 5.11. Class diagram of the *ontology* package

The ontologies used by the System are published in the Shared Memory. They have to be discovered and collected before their use. The class *OntologyManager* is in charge of the management of the ontologies. The method *discoverOntologies()* allows to search the ontologies and to collect them in the peer. The retrieved ontologies are represented through the class of type *Ontologies*.

The patterns are defined in the file *patterns.xml* contained in the folder *patterns*. The file *server.properties* contains the configuration properties that fix these parameters.

server.properties

```
...
patterns_directory_name=patterns
patterns_archive_name=patterns.xml
...
```

The class *PatternsManager* provides the functionalities for the management of the patterns. The method *loadPatterns()* loads the patterns from the XML file. During the indexing activities, at each step, it is necessary to run a SPARLQ query. The class *PatternsManager* provides the method *execQuery(...)* for running a query. The query is then sent to the class *OntologyManager*.

The patterns are structured as follows:

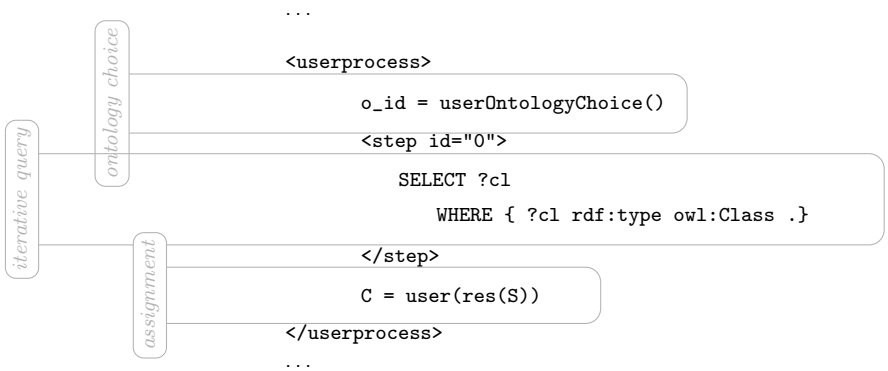
```
<patterns>
        <pattern>
                <id> ... </id>
                <name> ... </name>
                <description> ... </description>

                <descriptiontemplate>
                        ...
                </descriptiontemplate>

                <userprocess>
                        ...
                </userprocess>

                <algorithm>
                        ...
                </algorithm>
        </pattern>

        <pattern>
                ...
        </pattern>

        ...
</patterns>
```

The *id* is a unique identifier that distinguishes each pattern; the *name* is the string used to label the pattern; the *description* is a string that briefly describes the pattern. The *description template*, the *user process* and the *algorithm*, are defined as explained in section 4.5. Let's consider, for instance, the case of the indexing pattern on a concept (detailed in section 4.5.3).

The part corresponding to the *description template* contains the set of descriptions defined by the patterns. It is extended with parameters.

```
...
<descriptiontemplate>
       Individual: _:d
          Types: system:Document
          Facts: system:hasInterest <C>
</descriptiontemplate>
...
```

The *user process* is composed of an *ontology choice*, *iterative queries* and *assignments*. *Ontology choice* concerns users' choices of the ontology of domain for indexing. The *iterative queries* are SPARQL queries that the System runs for proposing the elements of the ontology necessary at the current step of the *user process*. The *assignments* are the choices users perform upon the proposed elements of the ontology. The *assignments* are used for fixing the parameters of the *description template*.



The *algorithm* is composed of the pseudo-code used in the procedure that creates the keys of indexing.

```
...
<algorithm>
       Key = {rdf : type,system : Document}
       append to Key {system : hasInterest,C}
</algorithm>
...
```

5.3.2 P2P

The *P2P* module provides functionalities for managing the infrastructure of Peers in the community. Its main implementation classes are under the package *p2parea*, depicted in the diagram of figure 5.12. The package is linked to three other packages: *sharedmemory*, used for indexing and storing of resources within the shared memory, *services*, used the web services that exploit the features of peers, and *freepastry*, that belongs to the FreePastry library and provides various low level functionalities of the P2P infrastructure.

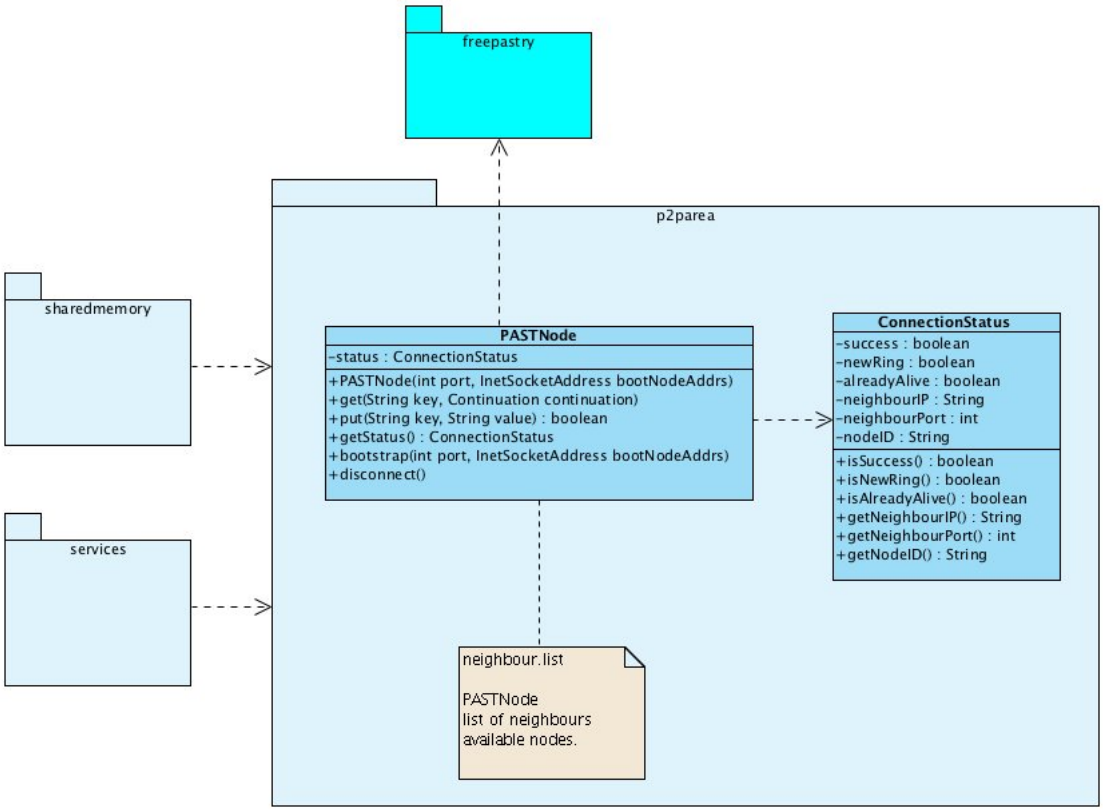


Figure 5.12. Class diagram of the *p2parea* and other linked packages

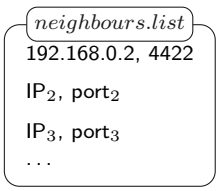
The main features of this layer are:

- to bootstrap a node;
- to manage the operations of publishing and retrieving within the distributed index.

A user interacts with the system through his or her user interface. The user interface (described in section 5.5) links the corresponding running node of the P2P

network and accesses the node's features through the *Services* layer (represented by the *services* package in figure 5.12). When a node runs, it links to other nodes of the P2P network. This step of a node's life is called *bootstrapping*. For bootstrapping, a node of the P2P network must know the existence of at least one other node, otherwise it cannot bootstrap or it is the first node of a new P2P network. For simplicity, let's imagine that we have a network composed of two nodes, *A* and *B*. The first time, the network is empty and no nodes are running. Node *A* starts and constitutes the first element of the P2P network. Node *A* is distinguished by its IP address, *ipA*, and its port, *portA*. When node *B* starts, it has to link to another node. Only one other node, node *A* is running, so node *B* starts because it knows the existence of the node *A*, namely *ipA* and *portA*. Node *B* joins the P2P network and is distinguished by its IP address, *ipB*, and its port *portB*.

The class *PASTNode* is created for bootstrapping a node and for performing publication and search on it. For bootstrapping, a *PASTNode* is fed the information of neighbours (IP and port), provided by the user interface through the tool called *P2P Connection* (further described in section **??**). If the link to the provided node is not possible, the *PASTNode* tries to link to one of the nodes with the given (IP an port) contained in the file *neighbour.list*. This file is part of the system. It is created manually at administrative level and provides a list of available nodes. The list is a collection of *(IP, port)* couples:



The *PASTNode* bootstraps through the method *bootstrap(...)*.

The user operates through the user interface that runs on all types of web browsers. The running peer, represented by an instance of a PASTNod, may be reached via the user interface at any time. The user interface is able to know the status of the corresponding peer because the *PASTNode* provides the status information represented by the class *ConnectionStatus*. This class contains information concerning the success of a bootstrapping, the possibility of managing an already alive node (that has been started before), the information concerning IPs and ports. This information is used through the *Services* layer.

For publishing an entry in the DHT, the *PASTNode* provides the method *put(...)*. The method gets the *key* of indexing and the *value* corresponding to the resource to index. The key is transformed to an hash code, as required by the FreePastry library. The process of publication progresses through the classes belonging to the FreePastry library.

For retrieving a resource, given a *key* of research, the *PASTNode* provides the method *get(...)*. This method gets the *key* of research and an object of type *Continuation*. For searching within a P2P network, it is necessary to run a query and to wait for the result, because they have to cross through some peers of the P2P network. The response to a query is not returned immediately. A Continuation is a listener (provided by the FreePastry library) on the conclusion of the operation of research in the network. When a response is available, the object of type Continuation is alerted and the returned data can be exploited.

Sometimes it may be necessary to disconnect the PASTNode, mainly for joining another peer in the P2P network. The disconnection is performed through the method *disconnect()*.

5.3.3 Memory

The *memory* module provides functionalities for storing and retrieving resources within the two memories. It is managed by two main packages: *sharedmemory* and *personalmemory*. Functionalities common to the two main packages are included in the package *memorycommon*.

5.3.3.1 Shared Memory

The *sharedmemory* package (see figure 5.13) contains classes for acting on the *Shared Memory*. The class *SharedMemoryManager* links the *p2parea* package for performing storage and retrieval of the Distributed Index. Its constructor gets an instance of the already created *PASTNode* that performs linking to the P2P network by the *SharedMemoryManager* class.

The *SharedMemoryManager* provides two main functionalities: publication and retrieval within the *Shared Memory*.

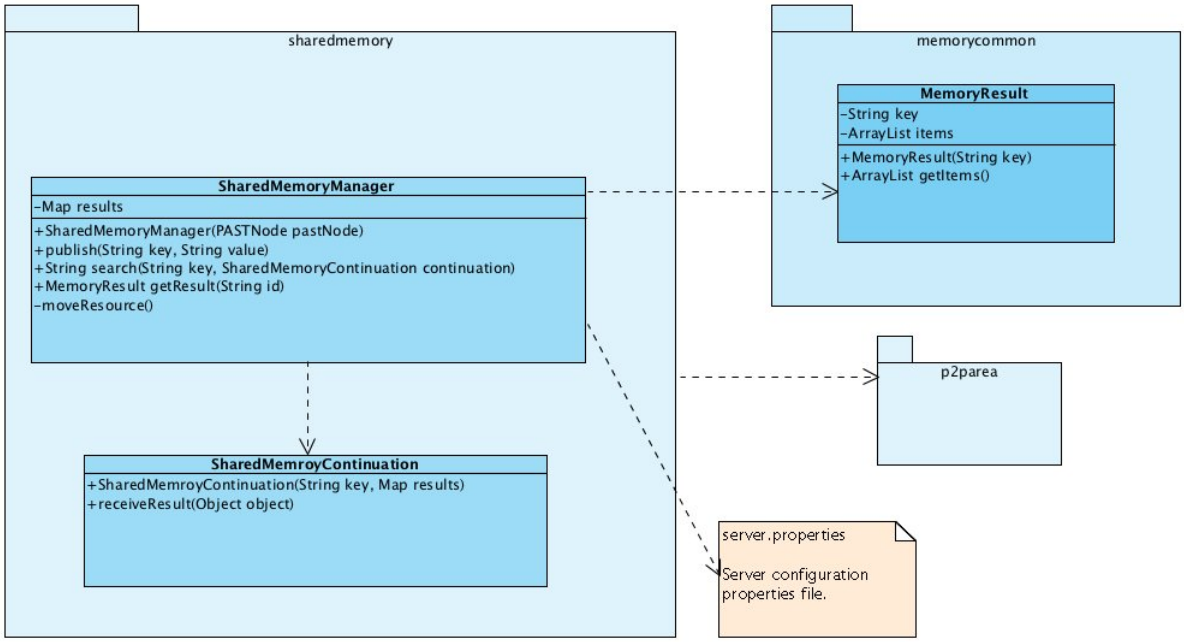


Figure 5.13. Class diagram of the *sharedmemory* package

Publishing a resource in the shared memory includes two activities:

- putting the entry related to the resource in the distributed index;
- physically saving the resource related to the entry in a peer's local archive.

The entry related to a resource to be published contains the key of indexing created for the resource. The key distinguishes between different types of resources or *documents* of *core elements* (ontologies, notes, wiki). For putting the entry in the distributed index, it is necessary to invoke the method *publish(...)* with the related *key* of indexing and the corresponding *value*.

The peer provides a temporary directory where the files corresponding to the resources to be published are saved before their publishing in the Shared Memory. The file *server.properties* contains the configuration properties that fix these parameters. The property *uploads_directory_name* corresponds to the temporary directory.

server.properties

```
...
shared_repository=docroot/sharedmemory
uploads_direcory_name=documents
personal_memory_directory_name=docroot/personalmemory
personal_memory_archive_name=personalmemory.info
...
```

If the resource is a document, the value is the reference of the resource in the local file system.

entry: (*key*, "/documents/file.ext")

For publishing the resource, it is necessary to copy the resource in the public folder of the peer (fixed by the property *shared_repository* in the property file). This is the folder where the local HTTP server gives access to shared resource. Before storing the entry, the value is substituted with the link to the local HTTP server.

entry: (*key*, "http://<url of the peer http server>/sharedmemory/file.ext")

If the resource is a *core element*, the value of the entry contains the entire resource content. For instance if the resource is a *Note*, the entry could be:

entry: (*key*, "Note content ...")

The wiki is a particular case because it is published only once at the administration level. No other activities of publishing are provided in this implementation of the system. The system allows to retrieve the wiki providing its specific key of research. Modifications in the wiki are possible by embedded functionalities.

For searching a resource with the key of research, the class *SharedMemoryManager* provides the method *search(...)*. This method gets the key of research and the *Continuation* as listener. The package *sharedmemory* defines a custom *Continuation* called *SharedMemoryContinuation*. The process of searching is redirected

to the instance of *PASTNode*. When the process is completed, the *SharedMemoryContinuation* is alerted through the method *receiveResult()*. Results are arranged within the data structure *result* of type *MemoryResult*. The results corresponding to each query are distinguished by the *key* of research. For exploiting the received data, it is necessary to invoke the method *getResult()*.

5.3.3.2 Personal Memory

The Personal Memory consits of a *local index* and a *local repository*. The *local index* is a file, local to the peer, that saves entries of the (key, value) type that correspond to the resources indexed by users in the Personal Memory. The *local repository* is the folder where the indexed resources are stored within the peer. The class *PersonalMemoryManager* manages the operations of storing and retrieving of resources within the Personal Memory.

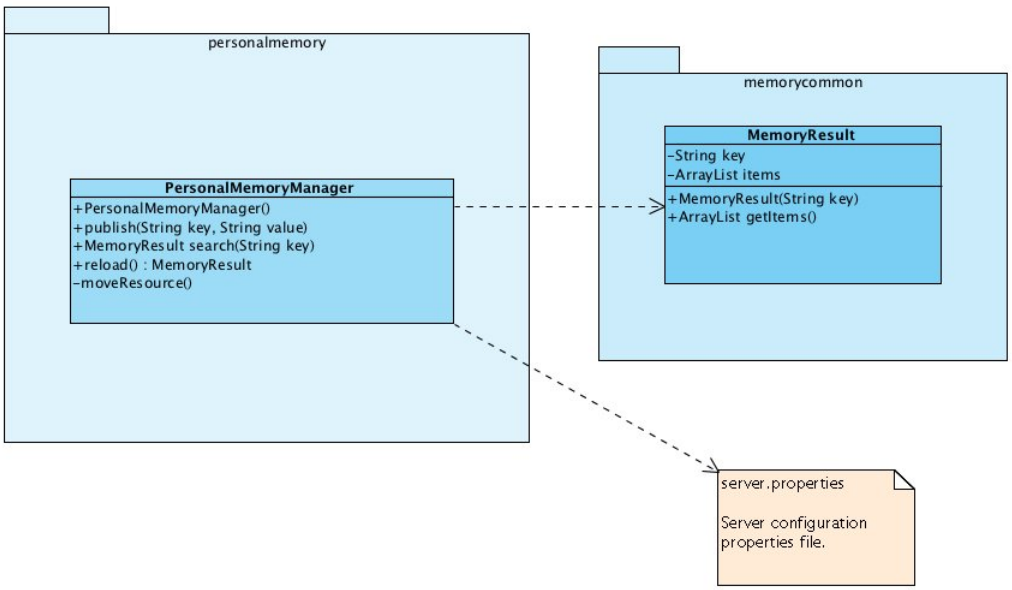


Figure 5.14. Class diagram of the *personalmemory* package

The configuration file *server.properties* fixes the parameters of the Personal Memory. The name of the *local index* file is given by the property *personal_memory_archive_name* (*personalmemory.info*). The *local repository* is stored in the folder *docroot/personalmemory*; its name is given by the property *personal_memory_directory_name*.

The Personal Memory is intended to contain resources of type *Document* or *Notes*. Even if there are no restrictions for storing other kinds of *core element* resources, we choose not to manage these kinds of resources in the Personal Memory.

For publishing a resource in the Personal Memory it is necessary to invoke the method *publish(...)* of the *PersonalMemoryManager* and give the related *key* of indexing and the corresponding *value*.

The content of the *personalmemory.info* file is composed of an undefined set of lines, where each line corresponds to an entry of the *local index*.

personalmemory.info

```
key1, http://<url of the peer http server>/personalmemory/file1.ext
key2, http://<url of the peer http server>/personalmemory/file2.ext
key3, "some content for a Note."
...
```

The *value* of an entry can be of two kinds: a content of a Note, or a reference to the file, if the resource is a document. The file is copied in the folder dedicated to the Personal Memory, in the local HTTP server, and it is accessed through its URL.

For searching a resource with a *key* of research, it is necessary to call the method *search(...)*. The process of research is accomplished simply by scanning the list of entries contained in the *personalmemory.info* file, and verifying a match between the key of research and the key of each entry. The result of searching are given through the object *MemoryResult*.

5.4 Services Layer

The *Services* Layer gives access to the functionalities of the underlying modules of the system through Web Services. The *services* package (see figure 5.15) contains the classes corresponding to four Web Services that implement the functionalities of the *Services* layer. It is divided into:

- *P2PWS*, that interacts with the *P2P* layer;
- *PersonalMemoryWS*, that gives access to the Personal Memory;
- *SharedMemoryWS*, that gives access to the Shared Memory
- *OntologyWS*, that relates the Ontologies and the Patterns defined in the System.

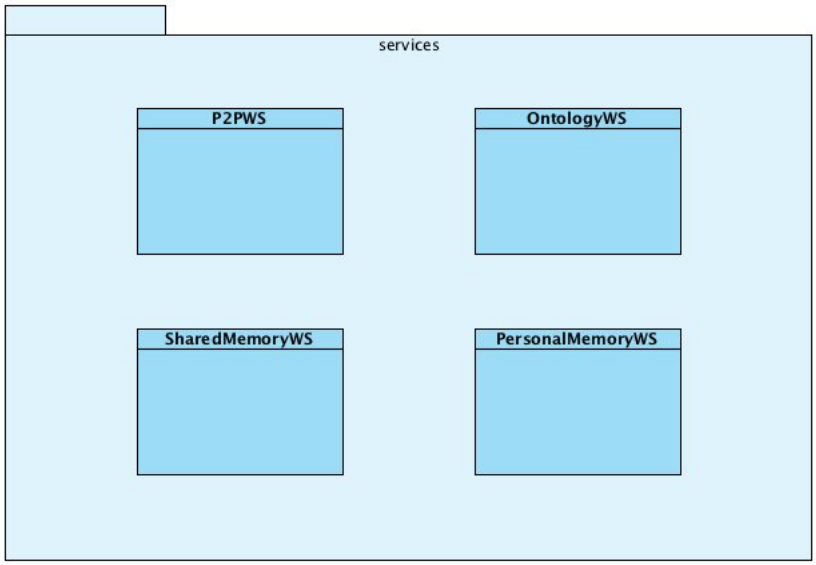


Figure 5.15. Class diagram of the *services* package

Web Services are designed following the REST (REpresentational State Transfer) architecture style. These Web Services are called RESTFul. The basic element of a Web Service is the resource (any information on the server side), intended as any item of interest. RESTful web services use HTTP protocol methods for the operations they perform. Table 5.1 shows the methods and their meanings. Resources can be accessed by clients using web URIs and HTTP:

http://www.utc.fr/resource/

Table 5.1. HTTP methods and the operations they perform

HTTP method	*Operation*
GET	Get a resource
POST	Create a resource
PUT	Update a resource
DELETE	Delete a resource

A representation of the resource is returned and the representation puts the client application into some state. The result of a new request by the client produces a new representation that places the client into yet another state. Thus, the client changes (transfers) a state with each resource representation. In our system RESTFul Web Services are developed through the JAX-RS Java API [20]. A RESTFul Web Service is associated to a resource class (a Java Class) using the *@Path* annotation (for Specification see [43]). The Web Service accessed by the URL *http://www.utc.fr/resource/* is associated to a class with the following annotation:

@Path("/resource")

The value of the annotation can have a relative URI path template in curly braces {...}. A URI path template acts as a placeholder for a relative path URI. Generally it is a string with zero or more embedded parameters in it, and it forms a valid URI path when the values are applied for the parameters. For instance, the path:

@Path("/resource/{id}")

could identify a resource accessible through the URL:

http://www.utc.fr/resource/53

intended as a request for a resource identified by the *id* number *53*.

Returned resource of Web Services, available in our system, are represented in JSON [3] format. JSON (JavaScript Object Notation) is a data format for interchange within client-server applications. It is a text format that is completely language independent. JSON data are expressed as collections of (name,value) pairs, or as ordered list of values. The following is an example representing a group of students:

[3] http://www.json.org

```
{
        "type": "group",
        "value": "students",
        "items": [
                {"name": "Giovanni", "age": "23"},
                {"name": "Luca", "age": "22"},
                {"name": "Matteo", "age": "25"}
        ]
}
```

5.4.1 P2PWS

Connections to the P2P network are performed through the P2PWS Web Service as links to the *P2P* layer. They are used by the peer of the P2P network to join/leave the network and to publish/search in the distributed index. User interaction is based on a Web user interface provided by a specific peer. The user can interact with the same peer any time from a Web browser located in any computer that is connected to Internet. A peer may be both connected to and disconnected from the P2P network. From the Web user interface, the user can connect to the P2P network by joining a peer that is already alive. It can be disconnected from a dead or corrupted peer and then connected again to another peer. The user is allowed to verify the status of an established connection. The resource class of the P2PWS is *services.P2PWS* associated to the path */p2p*.

Resource Class	services.P2PWS
Path	@Path("/p2p")

P2PWS Web Service is a container resource of three sub-resources: *bootstrap*, *status*, *disconnect*.

5.4.1.1 Bootstrap

For bootstrapping a node, the P2PWS Web Service provides a sub-resource that is accessible through the relative URI path template:

/p2p/bootstrap/{local_port}/{neighbour_ip} /{neighbour_port}

The table 5.2 provides the details concerning the *bootstrap* sub-resource. The URL necessary for accessing the web service is in the form:

http://www.utc.fr/p2p/bootstrap/4422 /192.168.0.3/2244

The parameters given in the URL are mapped to local variables of the class *services.P2PWS*.

The Web Service returns a JSON object that represents the status of the established connection or the failure message via the *"success", "no"* parameter. For creating the returned response, the system performs a transformation of an object of type *p2parea.ConnectionStatus* to a JSON text representation.

Table 5.2. Details of the Bootstrap sub-resource

Path	@Path("/p2p/bootstrap/{local_port}/{neighbour_ip} /{neighbour_port}")
URL	http://<host>:<port>/p2p/bootstrap/<local_port> /<neighbour_ip>/<neighbour_port>
Method	POST
Params	@QueryParam("local_port") String local_port, @QueryParam("neighbour_ip") String neighbour_ip, @QueryParam("neighbour_port") String neighbour_port,

Return	{
	"success" : "yes", // [yes, no]
	"new_ring" : "no", // [yes, no]
	"node_ip" : "...",
	"node_port" : "...",
	"node_id" : "...",
	"already_alive" : "...", // [yes, no]
	"neighbour_ip" : "...",
	"neighbour_port" : "..."
	}

The returned JSON object consists of:

- *success*, of the bootstrap operation;
- *new_ring*, if the node is the first node a new P2P network;
- *node_ip*, the IP address of the node;
- *node_port*, the port of the node;
- *node_id*, the id of the node; it is necessary in case of the disconnection is requested;

- *already_alive*, in the case that the node has been bootstrapped before;
- *neighbour_ip*, the IP address of the neighbour node;
- *neighbour_port*, the port of the neighbour node.

5.4.1.2 Status

The peer is accessible by users at any time through any Web browser. When the peer is reached, the system verifies whether the P2P node is already connected to the P2P network of the community. For that, the P2PWS Web Service provides the sub-resource represented by the path template

/p2p/status

(see details in table 5.3). The Web Service returns the status of the actual connection. If no connections is established, the user may decide to connect to the P2P network through the P2PWS Web Service.

Table 5.3. Details of the Status sub-resource

Path	@Path("/p2p/status")
URL	http://<host>:<port>/p2p/status
Method	GET
Params	*none*

Return	{
	"success" : "yes", // [yes, no]
	"new_ring" : "no", // [yes, no]
	"node_ip" : "...",
	"node_port" : "...",
	"node_id" : "...",
	"already_alive" : "...", // [yes, no]
	"neighbour_ip" : "...",
	"neighbour_port" : "..."
	}

5.4.1.3 Disconnect

For disconnecting a peer, the P2PWS Web Service provides the sub-resource *Disconnect* (detail in table 5.4). The information necessary for disconnecting a peer is its *node_id*. This information is provided when the peer has been bootstrapped, or when the status is requested.

Table 5.4. Details of the Disconnect sub-resource

Path	@Path(”/p2p/disconnect/{node_id}”)
URL	http://<host>:<port>/p2p/disconnect/<node_id >
Method	PUT
Params	@QueryParam(”node_id”) String node_id,

Return	{
	”success” : ”yes” // [yes, no]
	}

5.4.2 PersonalMemoryWS

The Personal Memory Web Service manages the operations of indexing, storing and retrieving in the personal memory. The content of the memory is stored locally in the peer. The index is contained in the file *personalmemory.info*. The archive of file resources is archived in the folder *personalmemory*.

Resource Class	services.PersonalMemoryWS
Path	@Path(”/personalmemory”)

The PersonalMemoryWS contains three sub-resources: *publish*, *search* and *reload*.

5.4.2.1 Publish

For publishing in the Personal Memory, the PersonalMemoryWS Web Service provides the sub-resource accessible through the relative URI path template

/personalmemory/publish

The data required for publishing are the *key* and the *value* that correspond to the entry to be published. The returned JSON object reports the success or failure information of the operation of publishing.

Table 5.5. Details of the Publish sub-resource

PATH	@Path(”/personalmemory/publish”)
URL	http://<host>:<port>/personalmemory/publish
Method	POST
Params	@FormParam(”key”) String key @FormParam(”value”) String value

Return	{
	”success” : ”yes” // [yes, no]
	}

5.4.2.2 Search

For searching it is necessary to provide the *key* of research. The relative URI path template

/personalmemory/search/{key}

requires to input the key when the PersonalMemoryWS Web Service is invoked for searching. The returned information is in the form of JSON data. The Web Service returns some information on the status of the research and, in case of success, the list of (key, value) pairs that correspond to the key of research. The key contained in the returned pairs cannot be equal to the key of research for the reason related to the key extension.

Table 5.6. Details of the Search sub-resource

PATH	@Path("/personalmemory/search/{key}")
URL	http://<host>:<port>/personalmemory/search/<key >
Method	GET
Params	@QueryParam("key") String key

Return	{
	success: "true", // [yes, no]
	id: "", //not used
	message: "", //in event of error
	results: [
	{ "key": "..." , "value": "..."},
	{ "key": "..." , "value": "..."}
	...
	]
	}

5.4.2.3 Reload

Sometimes it is useful to get the list of all data archived in the local index. The PersonalMemoryWS Web Service provides the relative URI path template

/personalmemory/reload

The returned information is the entire content of the local index.

Table 5.7. Details of the Reload sub-resource

PATH	@Path("/personalmemory/reload")
URL	http://<host>:<port>/personalmemory/reload
Method	GET
Params	*none*

Return	{
	success: "true", // [yes, no]
	id: "", //not used
	message: "", //in event of error
	results: [
	{ "key": "..." , "value": "..."},
	{ "key": "..." , "value": "..."}
	...
	]
	}

5.4.3 SharedMemoryWS

The Shared Memory Web Service manages the operations of indexing, storing and retrieving in the shared memory. The distributed index is scattered in the P2P network within a DHT.

Resource Class	services.SharedMemoryWS
Path	@Path("/sharedmemory")

The SharedMemoryWS contains three sub-resources: *publish*, *search* and *getresults*.

5.4.3.1 Publish

For publishing in the Shared Memory, the SharedMemoryWS Web Service provides a sub-resource accessible through the relative URI path template

/sharedmemory/publish

The required data are the key and the value that correspond to the entry to be published. The returned JSON object reports the success or failure information of the operation of publishing.

Table 5.8. Details of the Publish sub-resource

PATH	@Path("/sharedmemory/publish")
URL	http://<host>:<port>/sharedmemory/publish
Method	POST
Params	@FormParam("key") String key @FormParam("value") String value

Return	{
	"success" : "yes" // [yes, no]
	}

5.4.3.2 Search

For searching it is necessary to provide the key of research. The relative URI path template

/sharedmemory/search/key

requires to input the key when the SharedMemoryWS Web Service is invoked for searching. The process of research within the distributed index is related to the P2P network. The results of such a operation are not immediately available because messages are propagated along several peers of the network. For that reason, the SharedMemoryWS Web Service does not return any data of the index but only an *id* as an identifier of the invoked operation of searching. The *id* is unique for each operation. It is used successively for requesting retrieved results.

Table 5.9. Details of the Search sub-resource

PATH	@Path(”/sharedmemory/search/{key}”)
URL	http://<host>:<port>/sharedmemory/search/<key >
Method	GET
Params	@QueryParam(”key”) String key

Return	{
	success: ”true”, // [yes, no]
	id: ”id_1255680347670”,
	message: ”” //in event of error
	}

5.4.3.3 Get results

The retrieved results of a search operation can be requested from the SharedMemoryWS Web Service through the sub-resource with the relative URI path template

/sharedmemory/getresults/{id}

It is necessary to provide the *id* of a *search* operation invoked before. The Web Service returns some information concerning the status of the research and, in case of success, the list of (key, value) pairs that correspond to the key of research. The key contained in the returned pairs cannot be equal to the key of research for the reasons related to the key extension.

Table 5.10. Details of the Ger Results sub-resource

PATH	@Path(”/sharedmemory/getresults/{id}”)
URL	http://<host>:<port>/sharedmemorymemory/getresults /<id >
Method	GET
Params	@QueryParam(”id”) String id
Return	{ success: ”true”, // [yes, no] id: ””, message: ””, //in event of error results: [{ ”key”: ”...” , ”value”: ”...”}, { ”key”: ”...” , ”value”: ”...”} ...] }

5.4.4 OntologyWS

The OntologyWS Web Service allows users to access the functionalities of the *Ontology* module. It is linked to the classes *OntologyManager* and *PatternsManager*. The main features it provides concer the *patterns* defined in the System and the operations on the *ontologies* available for creating the keys of indexing. The patterns are shown to users thorough the Web user interface. Each pattern proposes a sequence of steps that the user can follow for creating the keys of indexing. At each step the OntologyWS Web Service runs a query, through the *OntologyManager*, for collecting the ontology elements required for the specific step.

Resource Class	services.OntologyWS
Path	@Path("/ontology")

The OntologyWS Web Service is a container resource of two sub-resources: *load* and *query*.

5.4.4.1 Load

For loading the indexing patterns defined in the System, the OntologyWS Web Service provides the sub-resource accessible through the relative URI path template

/ontology/load

The web service returns a JSON formatted data containing the set of indexing patterns, structured as follows:

- *id*, a unique identifier that distinguishes the pattern;
- *name*, the string used to label the pattern;
- *description*, a brief descriptive statement for the pattern;
- *descriptiontemplate*, contains the set of descriptions defined by the pattern;
- *userprocess*, defines the process of interaction with the user;
- *algorithm*, the pseudo-code used for creating the keys of indexing.

Table 5.11. Details of the Load sub-resource

PATH	@Path("/ontology/load")
URL	http://<host>:<port>/ontology/load
Method	GET
Params	none
Return	{ "patterns": [{ "id": "", "name": "", "description": "", "descriptiontemplate": "", "userprocess": "", "algorithm": "" }, { ... }] }

5.4.4.2 Query

An indexing pattern is articulated in several steps. At each step, the System collects from the chosen ontology the parts that are necessary to users for selecting an ontological element related to the current step. User's selections are required at each step for the following step to happen. For querying the chosen ontology at each step of the indexing pattern, the OntologyWS Web Service defines a sub-resource relative to the URI path template

/ontology/query/{pattern_id}/{step}/{ontology_id}/{query}

To invoke such a sub-resource it is necessary to provide the following parameters:

- *pattern_id*, the identifier of the indexing pattern;
- *step*, the number of the step;
- *ontology_id*, the identifier of the chosen ontology;

- *query*, the SPARQL query to execute.

The web service returns a JSON object that contains the *success* and *message* information. The *results* are returned as a set of items with the name of the variable in the SPARQL query and the relative values.

Table 5.12. Details of the Query sub-resource

PATH	@Path(”/ontology/query/{pattern_id}/{step} /{ontology_id}/{query}”)
URL	http://<host>:<port>/ontology/query/<pattern_id>/ <step>/<ontology_id >/<query>
Method	GET
Params	@QueryParam(”pattern_id”) String pattern_id @QueryParam(”step”) String step
Params	@QueryParam(”ontology_id”) String ontology_id
Params	@QueryParam(”query”) String query

Return	{
	success: ”yes”, // [yes, no]
	message: ””, //in event of error
	results: {
	”head”: {
	”vars”: [”cl”, ...]
	},
	”items”: [
	{
	”cl”: { ”type”: ”...”, ”value”: ”...”},
	...
	}
	]
	}
	}

5.5 Front-end Layer

The *Front-end* layer provides the Web User Interface that allows users to interact with the system. The figure 5.16 shows the diagram of the *Front-end* module, containing the development of this layer.

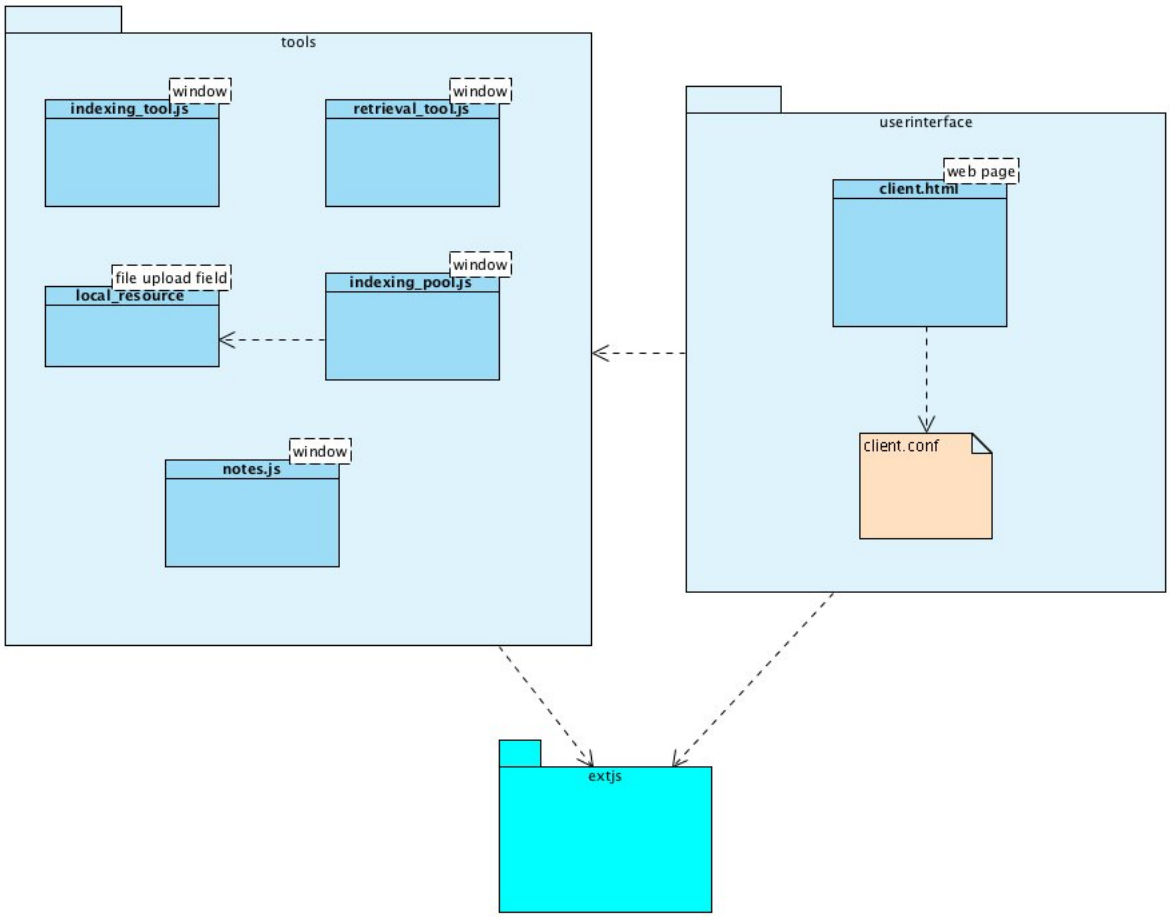


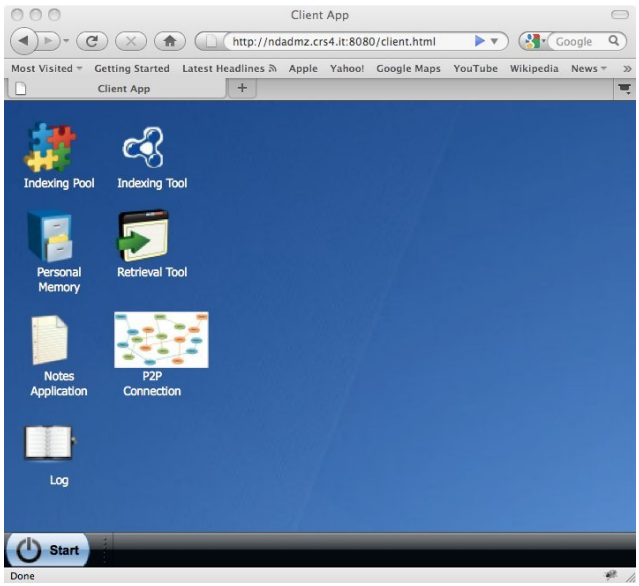
Figure 5.16. The diagram of the Front-end

The package *userinterface* contains the main HTML file *client.html*. It defines the design of the Web User Interface (hereafter UI) and is based on mixing the technologies HTML, JavaScript and CSS. The package links the JavaScript library ExtJs2, included in the package *extjs*. The file *client.html* needs a set of external JavaScript files defined in the package *tools*. This package contains the tools used within the UI. Such tools implement the several functionalities of the UI. The file *client.html* also requires the file *client.conf* containing the configuration settings.

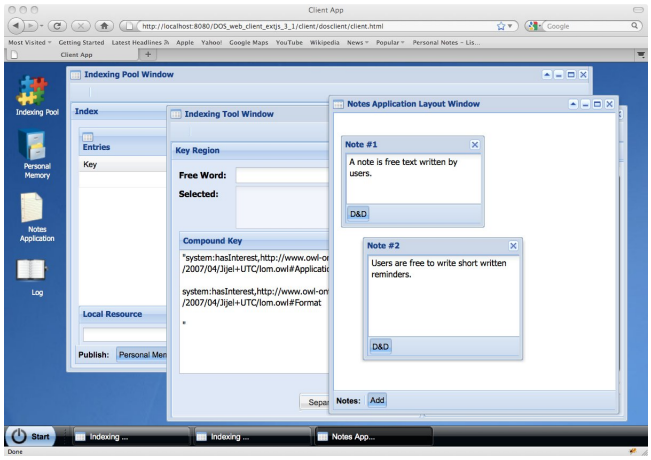
5.5.1 User Interface

The UI is reachable through the URL referred to the peer where it is installed. For instance the following URL refers to a UI:

http://ndadmz.crs4.it:8080/client.html



(a)



(b)

Figure 5.17. The Web user interface

The UI is created in the shape of a traditional Desktop of an Operating System (see figure 5.17 (a)). It contains a set of icons corresponding to the tools the users

may run. Each tool may be opened by clicking at the corresponding icon. In figure 5.17 (b) is shown the user interface with certain tools opened.

5.5.1.1 Configuration

The UI needs the settings of some configuration parameters. The parameters are contained in the file *client.conf* and refer to the URI of the web services invoked for exploiting the functionalities of the several tools.

client.conf

```
'server_name' : 'http://' + window.location.hostname,
'server_port' : '8080',
'ws_p2p_bootstrap' : '/p2p/bootstrap',
'ws_p2p_status' : '/p2p/status',
'ws_p2p_disconnect' : '/p2p/disconnect',
'ws_personalmemory_publish' : '/personalmemory/publish',
'ws_personalmemory_search' : '/personalmemory/search',
'ws_personalmemory_get' : '/personalmemory/reload',
'ws_sharedmemory_publish' : '/sharedmemory/publish',
'ws_sharedmemory_search' : '/sharedmemory/search',
'ws_sharedmemory_getresults' : '/sharedmemory/getresults',
'ws_ontology_load' : '/ontology/load',
'ws_ontology_query' : '/ontology/query',
```

5.5.2 Tools

The UI provides a set of tools: *Indexing Tool*, *Indexing Pool*, *Local Resource*, *Notes*, *Retrieval Tool*. The set of tools can be extended for improving the UI with new functionalities for different purposes.

5.5.2.1 Indexing Tool

The *Indexing Tool* is used for choosing the ontologies retrieved from the network and for creating the keys of indexing. For creating a key of indexing, the *Indexing Tool* and its indexing patterns allow to browse the ontologies selected by the user such that only their currently relevant parts are displayed. The figure 5.18 shows a view of the *Indexing Tool* where through the *Iterative Pattern* the key shown in the *Compund Key* box is created.

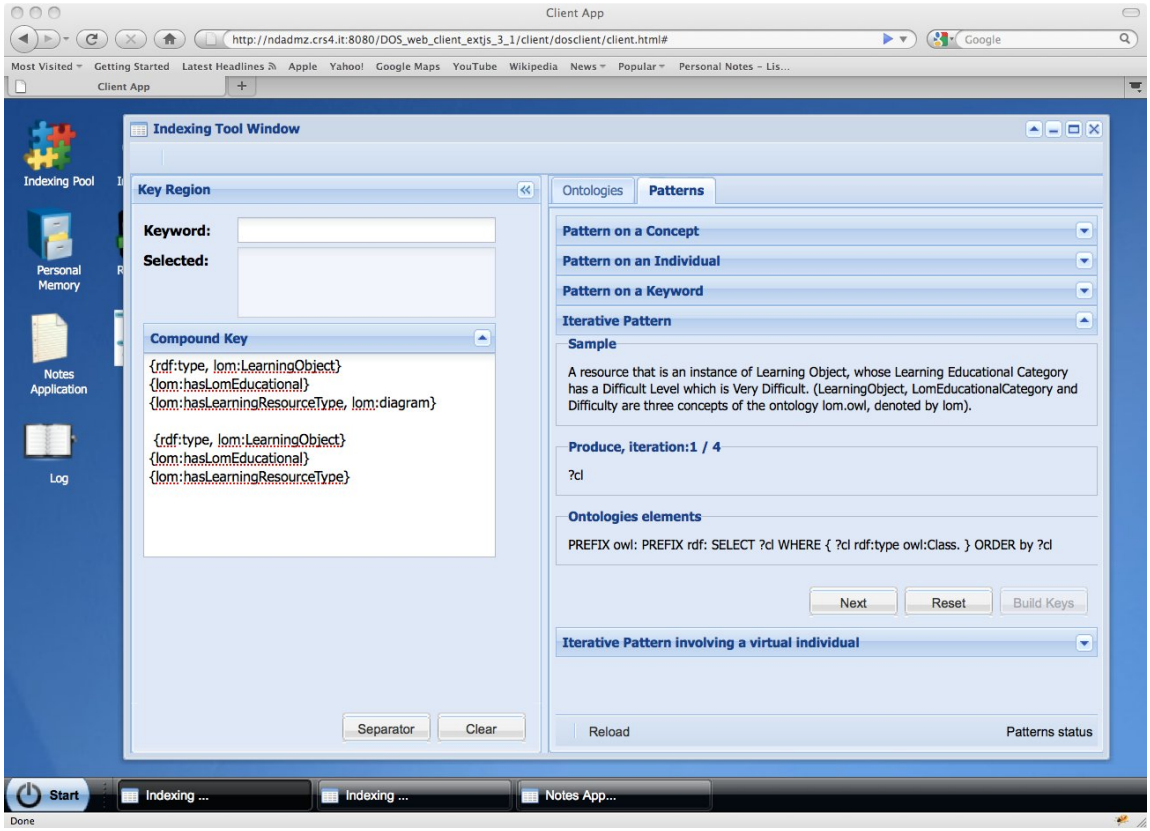


Figure 5.18. The Indexing Tool

For associating a key of indexing to a resource, it is necessary to use the Indexing Tool and the Indexing Pool. A simple drag and drop action from the first to the second, enables to link the created key to a resource selected through the Indexing Pool.

5.5.2.2 Indexing Pool

The *Indexing Pool* is a temporary container of (key, resource) pairs. The resources may be published in both Shared Memory and Personal Memory.

The Indexing Pool allows users to select the resource they want to index and to associate the key of indexing built with the Indexing Tool.

After select the resource they want to index, the selected resource is associated to the key of indexing created through the indexing tool. The same key can identify several resources. The key may be used either to publish the resource in the Personal Memory or in the Shared Memory. The figure 5.19 shows the *Indexing Pool* where seven resources are associated to seven keys of indexing.

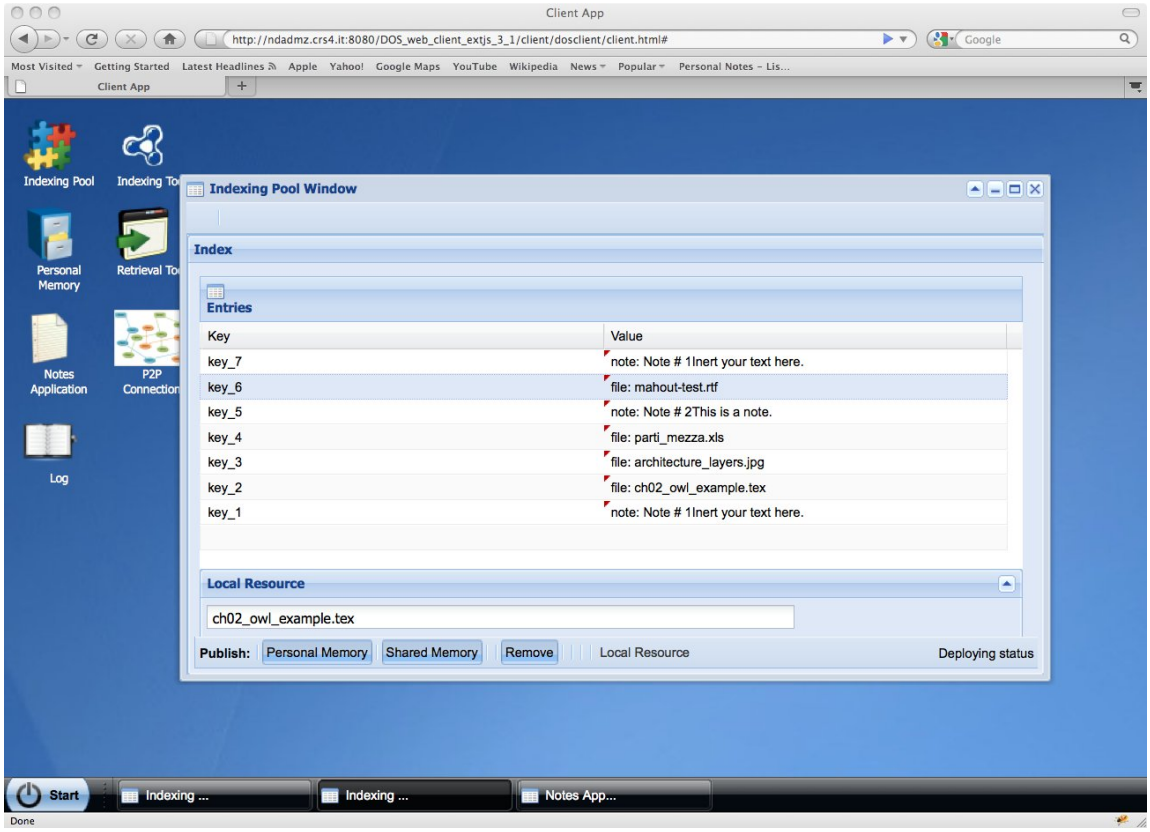


Figure 5.19. The Indexing Pool

5.5.2.3 Local Resource

The *Local Resource* tool allows to select resources from the local file system. Such resources can be uploaded to the server in order to proceed with their publication.

5.5.2.4 Notes

The *Notes* is a tool that enables users to create personal notes. The *Notes* may be associated to keys of indexing and published. The figure 5.20 shows the *Notes* tool with the notes created by the user. The notes are moved to the *Indexing Pool* for their publication.

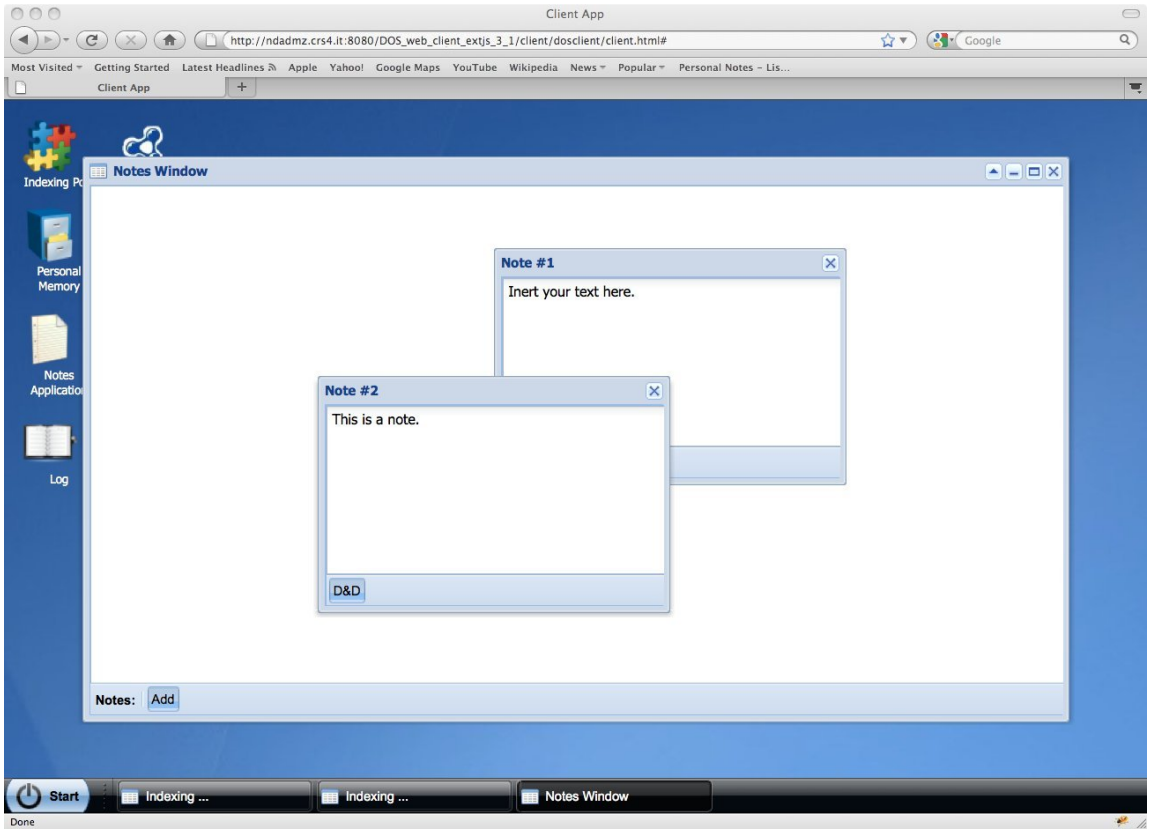


Figure 5.20. The Notes tool

5.5.2.5 Retrieval Tool

The *Retrieval Tool* allows users to submit queries to the system. It retrieves results and displayes them.

To retrieve a resource, it is necessary to create a key of research. The process is analogical to the one described for publishing. As soon as the key is completed, the Retrieval Tool launches a request to the Personal Memory and/or to the Shared Memory.

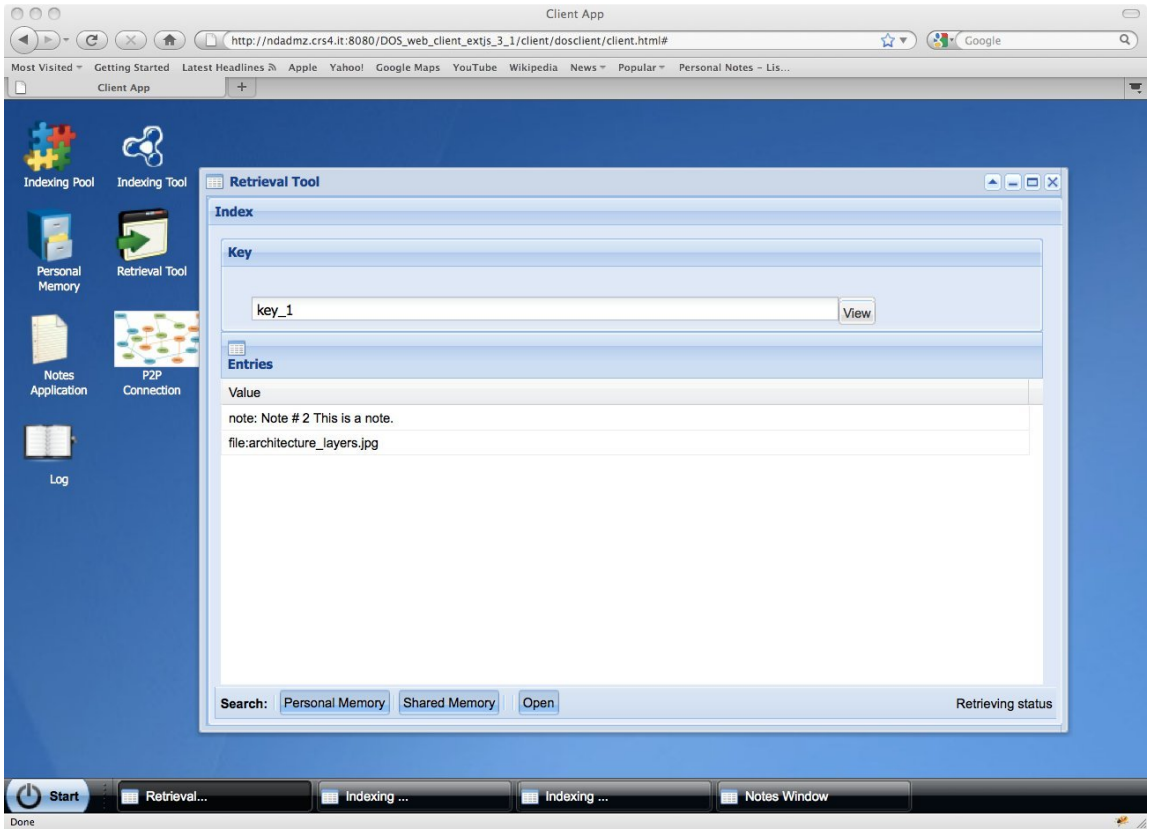


Figure 5.21. The Retrieval tool

Chapter 6

Experimentation

In this chapter, we describe the experiments performed on the System for testing our approach. We specify the equipment and the configuration needed to execute the experiments. We explain the test set of entries, created for using with various Communities or peers. Finally, we discuss and compare the results of the experiments.

6.1 Introduction

We executed some tests on the System in order to validate our approach and to collect important information on its performance. For testing the approach, we used a test set of entries that we created ad hoc, including the ontologies described in section 4.1.3.1. We based our experiments on different Communities of P2P nodes. Each test consisted in

- publishing the entries ((key, value) pairs) of the test set in the System;
- querying the System using the keys of the test set;
- calculating the number of successful and failed publications;
- calculating the number of successful and failed retrievals;
- calculating the time needed for retrieving the results.

6.2 Requirements

These experiments may be executed on various operating systems: Microsoft Windows, Linux, Mac OS. There are no particular minimal hardware requirements or

restrictions. It is necessary to configure the system with the Java Virtual Machine version 6 or higher.

6.3 Test set of entries

The test set is a collection of entries composed of a group of keys of indexing and a selection of 100 resources choosen from a local repository. For simplicity, such resources have been renamed as *Resource_1*, *Resource_2*, ..., *Resource_100*.

The test set is represented within an XML file named *entries.xml*. The file is structured as follows:

```
<entries>
    <entry>
        <key>
            <item> ... </item>
            <item> ... </item>
            ...
        </key>
        <value> ... </value>
    </entry>
    ...
</entries>
```

The tag `<entry>` corresponds to a (key, value) pair. Since a key may be composed of several items, the tag `<key>` contains a list of tags `<item>`. The tag `<value>` represents the resource. A key composed of several items will generate a list of (key, value) pairs, related to the same *value*. Every (key, value) pair is published individually.

For creating the keys of indexing we have used the *system.owl* ontology (with namespace prefix *system*) and the set of 5 ontologies described in section 4.1.3.1, summarized in the following table:

Table 6.1. Set of ontologies

Ontology file	*namespace prefix*
lom.owl	lom
lt.owl	lt
tg-release3-1.owl	tg
foaf.owl	foaf
GeoSkills.owl	geo

We consider the 7 cases of indexing (see section 4.3.4) and the respective extensions. The next table shows the number of extensions for each case of indexing:

Table 6.2. Cases of indexing

Case	*Extensions*
Concept	none
Property	none
Individual	1
Keyword	none
Virtual individual	2
Iterative	1
Iterative + virtual individual	2

For each case of indexing, we created at most 10 keys (with respective extended keys) for each ontology. There are 300 different keys. Considering the set of 100 resources, we assigned each key to 3 resources (3 keys / resource).

The following is an excerpt of the file *entries.xml*:

```
<entries>
	<entry>
		<key>
			<item> {rdf:type,system:Document}{system:hasInterest, lom:Axiom_1} </item>
		</key>
		<value> Resource_1 </value>
	</entry>
	...
	<entry>
		<key>
			<item> {rdf:type,system:Document}{system:hasInterest, lom:idea} </item>
			<item> {rdf:type,system:Document}{system:hasInterest, lom:Purpose} </item>
		</key>
		<value> Resource_10 </value>
	</entry>
	...
	<entry>
		<key>
			<item> {rdf:type,system:Document}{system:hasInterest, geo:Reduce}
				{geo:commonName, 'Minimal terms'} </item>
			<item> {rdf:type,system:Document}{system:hasInterest, geo:Reduce} </item>
			<item>{rdf:type,system:Document}{system:hasKeyword, 'Minimal terms'} </item>
		</key>
		<value> Resource_89 </value>
	</entry>
	...
</entries>
```

For simplicity reasons, we use the following namespace prefixes (abbreviations) in the examples: *rdf*, *system*, *lom*, *geo*, etc. During the experiments, the abbreviated namespace prefixes are substituted by their full names.

The test set is composed of only simple keys (these keys refer to simple descriptions, 4.2.4). Complex keys (see complex description 4.2.5) are processed by simple keys combined with the AND operator (see section 4.1.4.9). Queries by complex keys generate multiple queries, one for each component simple key. The results are collected together and the final result set is the intersection of their results. We do not consider complex keys because the response time of a complex query is the sum of the response times of the simple compounding queries.

6.4 Test environment

The test environment is created from the parts of the System able to manage a P2P node and its features. It provides a script for each of the following functionalities:

- running a P2P node;
- running the publication;
- running the search.

For running a P2P node, it is necessary to provide the information about neighbours available in the community. The first node that starts a P2P network is an exception because it does not join any neighbours.

The script in charge of publishing the entries reads the file *entries.xml*. For each *item* of each *entry*, the (item, value) pair is published in the DHT. A *log* file is created for saving some information that results from the publication process. The *log* file reports the success/failure message and the elapsed time of each publication. It is possible to verify the quota of DHT stored in each peer. When a new entry is published in the DHT, the System creates a unique identifier (composed of 160 bits). It is used for distinguishing the entry. The peer that saves the entry, creates a new file named with the unique identifier. Since the entry is replicated, the same file is created in more peers. It is possible to verify which entries are stored in a particular peer by looking at the list of such files.

For searching within the DHT, the script reads the file *entries.xml* and for each *key* runs a query. A *log* file saves the information related to this process. It reports the message of successes and failures of the search operation. For each key of research, the list of associated resources and the time necessary for getting the results is reported.

6.5 Running the Tests

The following tests were executed creating different configurations of Communities. We created Communities in a Local Area Network (LAN) exploiting both one single computer and multiple computers. Moreover, we used a computer located in a *demilitarized zone* (DMZ) and computers connected by ADSL.

In every organization a LAN is firewalled and protected from unsolicited external access (in particular from Internet). For security reasons, an access to a LAN is allowed only for trusted connections. A DMZ is a subnetwork (the segment of a LAN) that exposes some services to the external access.

An *ADSL* connection to Internet is a form of connections performed through a modem that uses an Internet Service Provider (ISP).

In every test we measured the number of successes of both published and retrieved resources.

We measured the time needed for publishing and retrieving the resources.

After that, we disconnected some nodes in order to verify whether there was any loss of data.

6.5.1 A Community of Multiple Nodes on the Same Computer

The first test involves a Community of 10 nodes running on the same computer: *node_1*, *node_2*, ... *node_10*. The *node_1* runs first, creating a new Community. The other nodes are created immediately after. The next figure shows the topology of the P2P network.

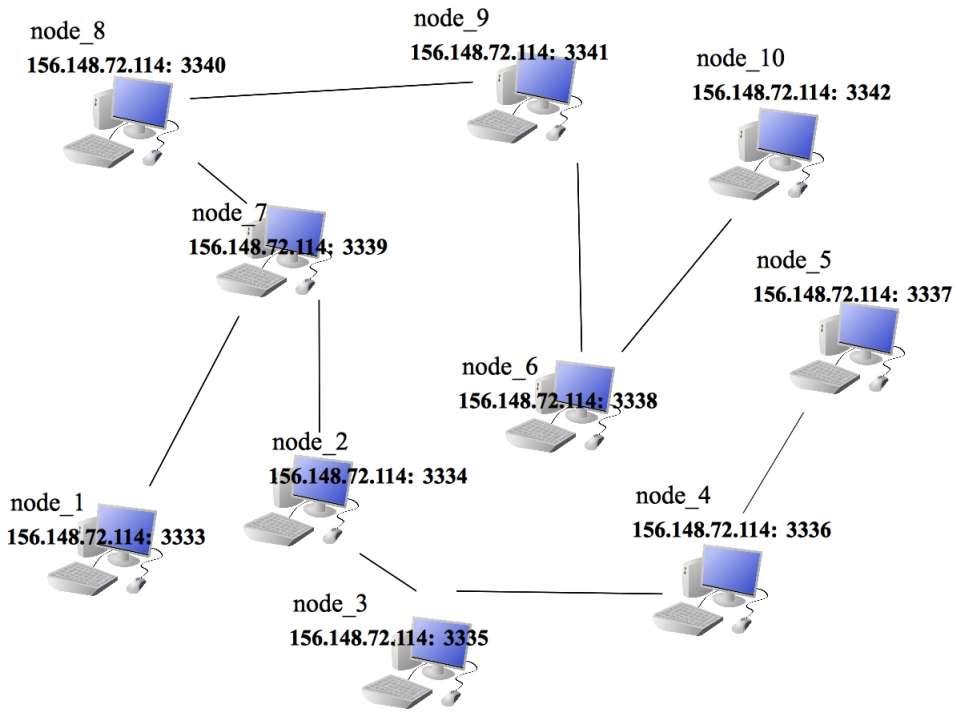


Figure 6.1. Topology of P2P network of 10 nodes on the same computer

The *node_9* runs the publication script. Publishing the 300 entries contained in the *entries.xml* file, generates 550 entries. There is no failure during the publication process. The significant times for publishing the entries are the following:

Minimum Publication Time: 9 ms.

Maximum Publication Time: 228 ms.
Average Publication Time: 21 ms.

The figure 6.2 shows a chart that reports the publication times for the 550 entries. In this chart and in the one presented after that, the x-axis reports the sequence of entries, while the y-axis reports time.

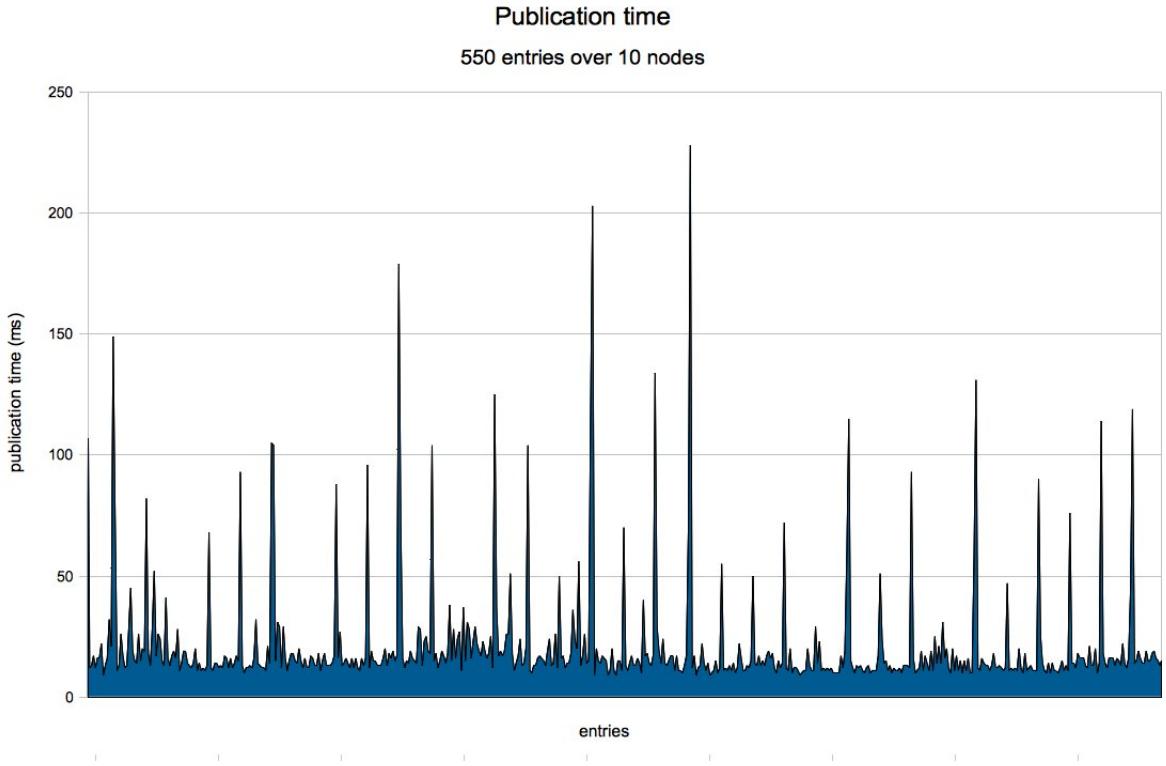


Figure 6.2. Publication times of 550 entries over 10 nodes on the same computer

Most entries require less than 50 ms for publishing.

The entries are distributed (quota of DHT) among the 10 peers. Moreover, each peer stores some replication of the entries. The following table shows the quota of DHT among the peers:

Table 6.3. DHT among 10 peers

Peer	*entries*
node_1	184
node_2	206
node_3	200
node_4	111
node_5	195
node_6	135
node_7	225
node_8	196
node_9	209
node_10	183

The sum of all the entries is 1.844 (including replicas), on average there are 184 entries per peer. Considering that we published 550 entries on 10 peers, there are 55 entries per peer. It means that each peer stores 3 replicas of other entries. An entry is replicated on three other peers.

The *node_10* runs the search script. Running the search script resulted in no failures. The significant times for retrieving the results are the following:

Minimum Retrieval Time: 1 ms.

Maximum Retrieval Time: 239 ms.

Average Retrieval Time: 5 ms

The chart reported in figure 6.6 shows search times for the 550 entries.

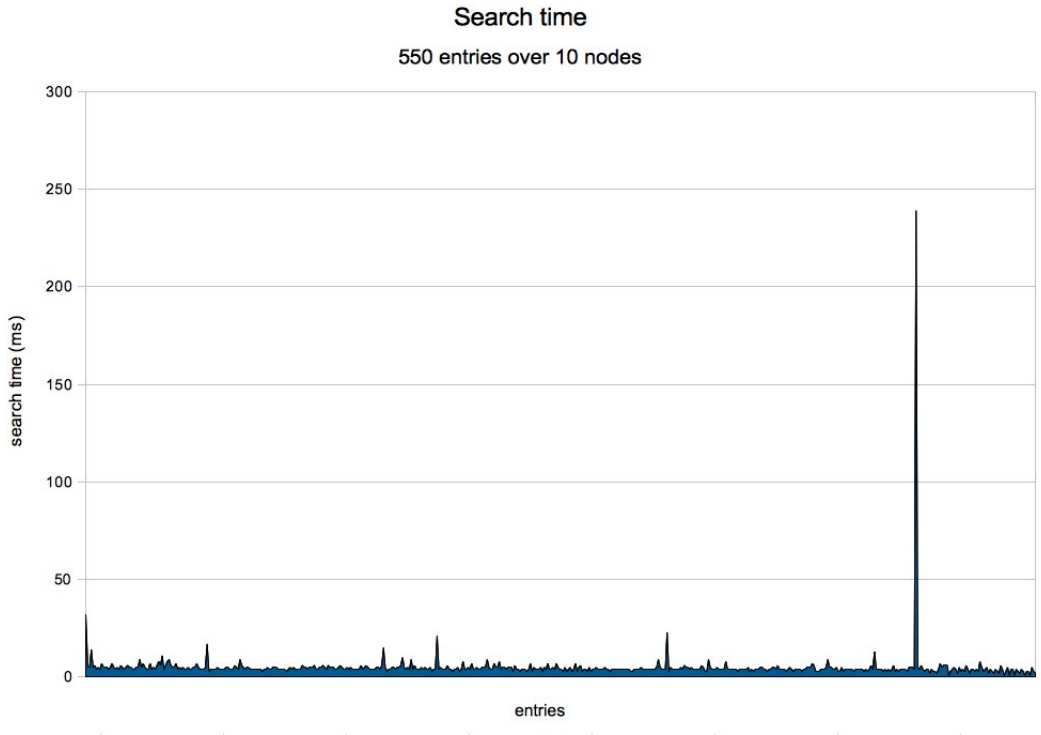


Figure 6.3. Search times of 550 entries over 10 nodes on the same computer

The search times are all under 50 ms, except onw case where the response time is 239 ms. It may have depended on the bandwidth available when the test ran. Even so, a longer response time is not problematic because query responses are collected asynchronously.

6.5.2 A Community of Nodes on Several Computers

The second test involves a Community of 7 nodes scattered on several computers: *node_1*, *node_2*, ... *node_7*. The *node_1* runs first, creating a new community. It is located in DMZ, even though in this configuration the feature was not exploited because the other nodes that were created immediately after, were located in the same LAN. Nevertheless, it was interesting to investigate whether there is a decrease in the performance that depended on the DMZ. The next figure shows the topology of the P2P network.

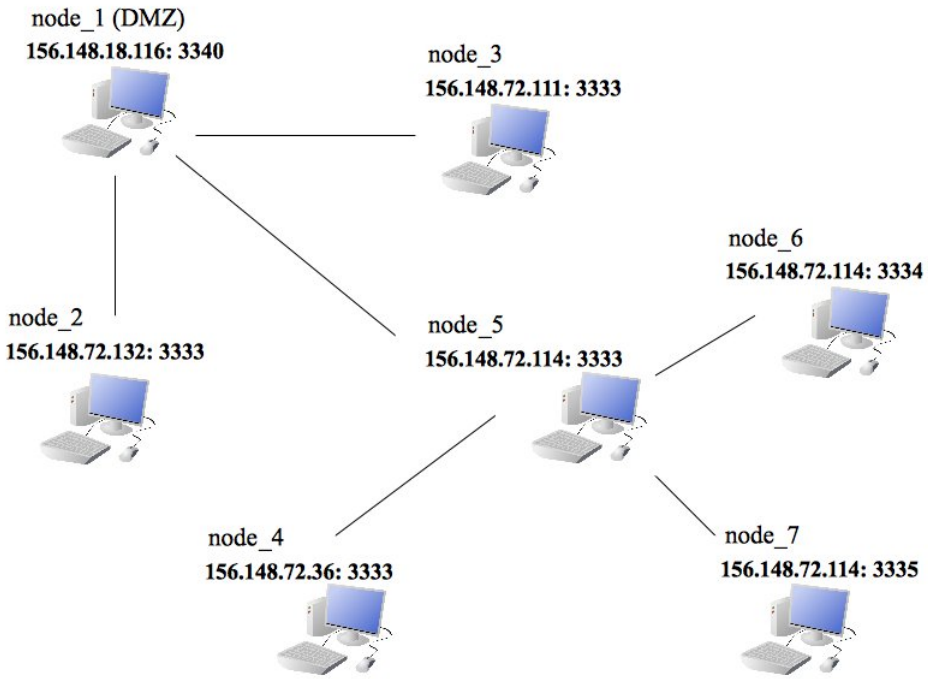


Figure 6.4. Topology of P2P network of 7 nodes on several computers

The *node_6* runs the publication script. There is no failure during the publication process. The significant times for publishing the entries are the following:

Minimum Publication Time: 4 ms.

Maximum Publication Time: 105 ms.

Average Publication Time: 13 ms.

The figure 6.8 shows the chart of publication times for the 550 entries.

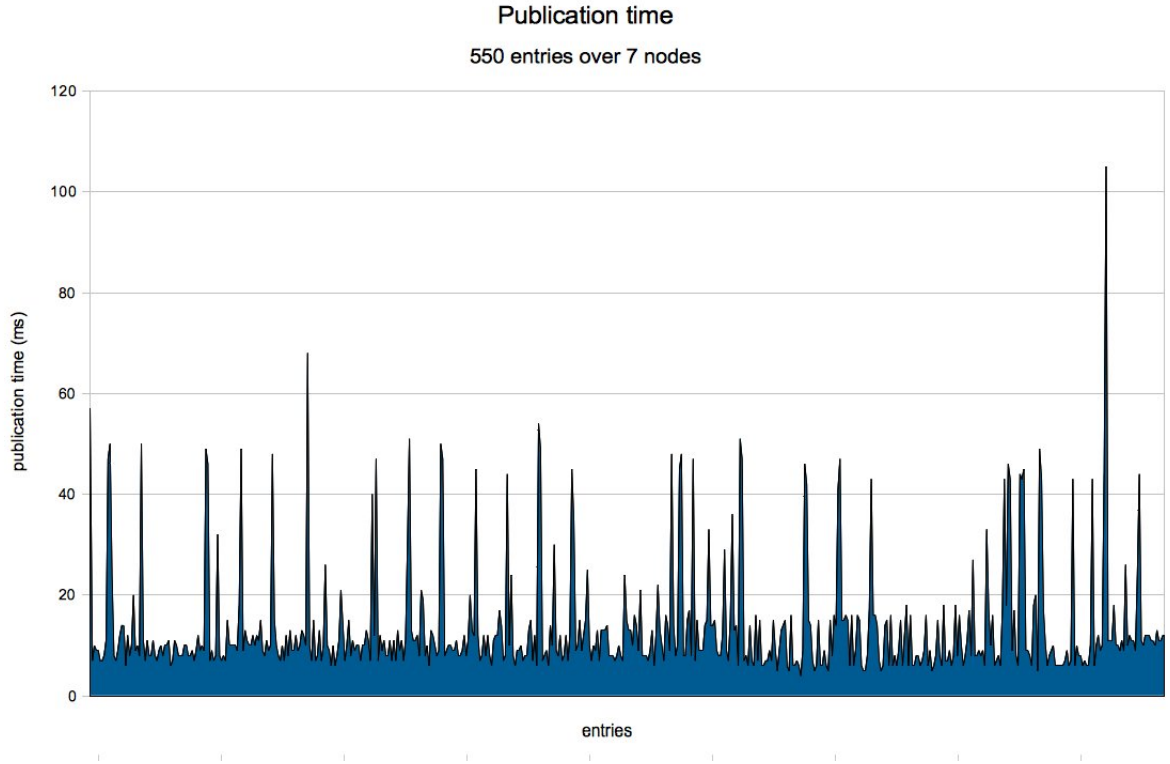


Figure 6.5. Publication times of 550 entries over 7 nodes on 5 computers

The distribution of entries (quota of DHT) among the 7 peers is as follows:

Table 6.4. DHT among 7 peers

Peer	*entries*
node_1	277
node_2	273
node_3	311
node_4	250
node_5	209
node_6	333
node_7	250

The sum of all the entries is 1.903 (including replicas), so on average there are 271 entries per peer. Considering that we published 550 entries on 7 peers, the distribution is 78 entries per peer. So, each peer stores 3 replicas of other entries and an entry is replicated on three other peers.

The *node_7* runs the searching script. Running the search script there are no failures. The significant times for getting the results are the following:

Minimum Retrieval Time: 1 ms.

Maximum Retrieval Time: 3008 ms.

Average Retrieval Time: 222 ms

The figure 6.6 shows the chart of search times for the 550 entries.

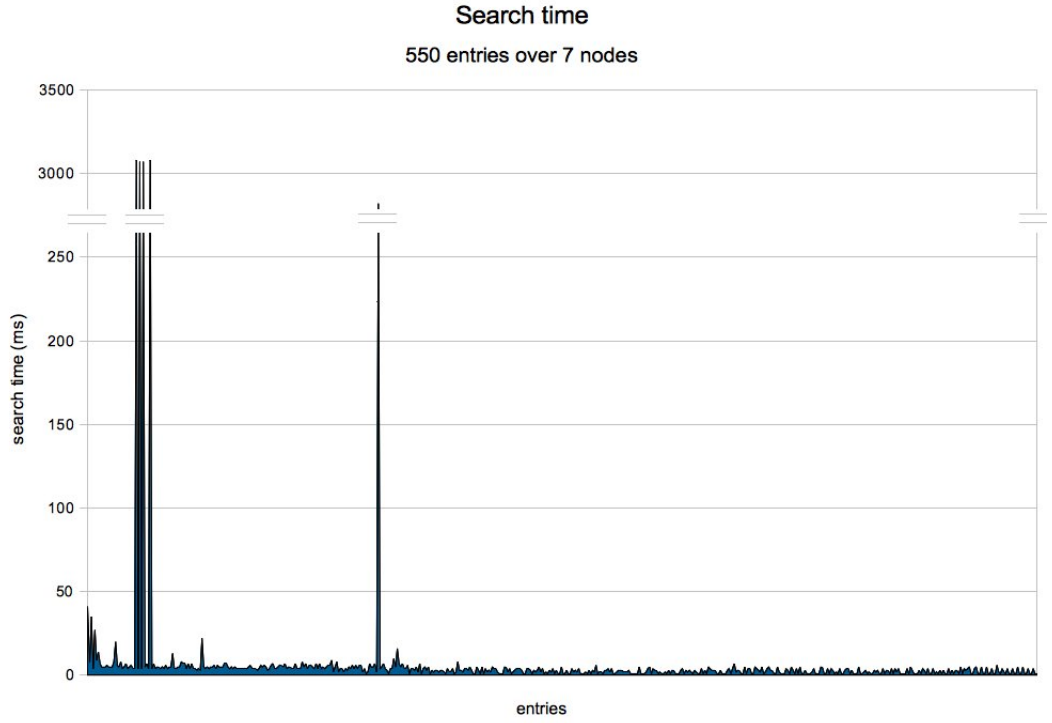


Figure 6.6. Search times of 550 entries over 7 nodes on 5 computers

There are some cases when the time needed to retrieve the results is higher than 3000 ms but the number of these cases is not significant.

6.5.3 A Community Involving Nodes Connected to Internet via ADSL

In this test, a Community of 5 nodes is scattered on several computers. The *node_1* runs first, creating a new community. It is located in DMZ. The other nodes start from computers connected to Internet via ADSL. In this case, it is fundamental to have the *node_1* located in DMZ. It allows the other nodes to join the Community. The next figure shows the topology of the P2P network.

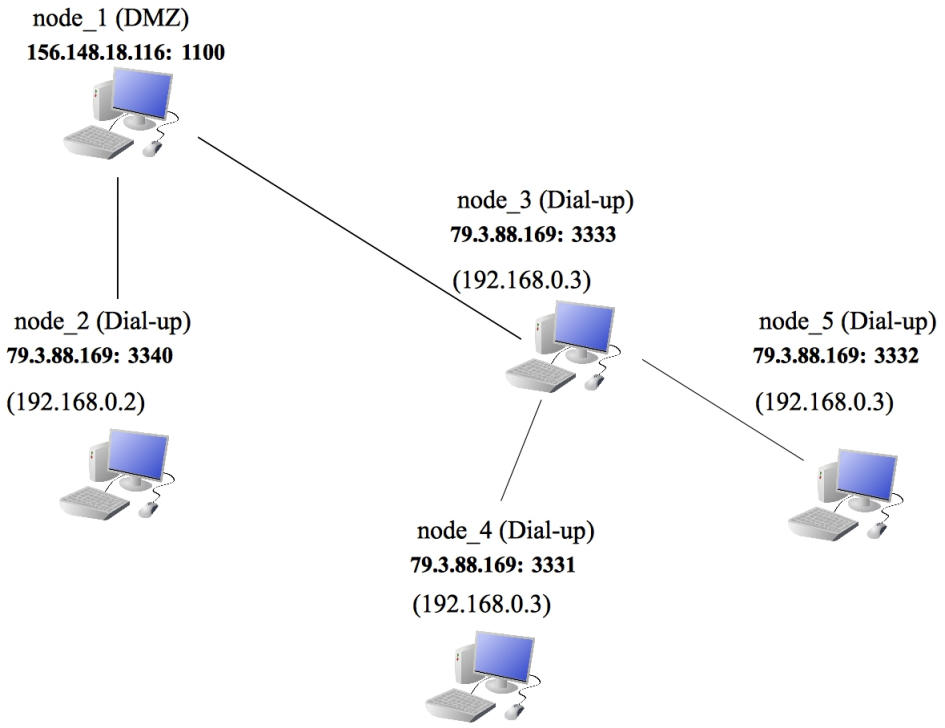


Figure 6.7. Topology of P2P network of 5 nodes on computers connected via ADSL

The connection to Internet takes place through a router with an ADSL modem (ADSL is the faster data communications technology over the telephone line). The router receives from the ISP the Internet Protocol 79.3.88.169. Two computers are connected to the router, creating a small LAN. The router gives one local IP address (192.168.0.2 and 192.168.0.3) to each. They are addressed by the IP of the router on the Internet.

The *node_4* runs the publication script. There is no failure during the publication process. The significant times for publishing the entries are the following:

Minimum Publication Time: 5ms.

Maximum Publication Time: 104ms.

Average Publication Time: 10ms.

The figure 6.8 shows the chart of the publication times for the 550 entries.

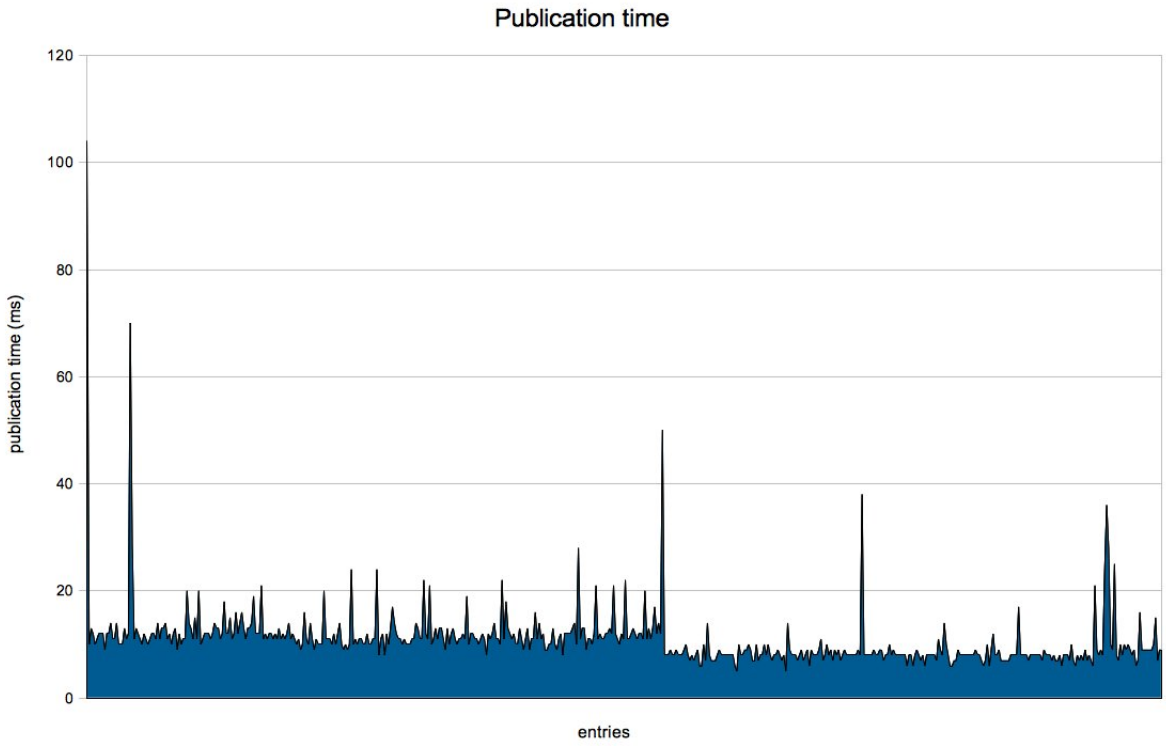


Figure 6.8. Publication times of 550 entries over 5 nodes on computers connected via ADSL

The distribution of entries (quota of DHT) among the 5 peers is the following:

Table 6.5. DHT among 5 peers

Peer	*entries*
node_1	359
node_2	305
node_3	305
node_4	436
node_5	306

The sum of all the entries is 1.711 (including replicas), then on average, there are 342 entries per peer. Considering we have published 550 entries on 5 peers, the distribution is 110 entries/peer. Also in this case, each peer stores 3 replicas of other entries.

The *node_5* runs the search script. There are some failures in running the search script: we get 531 responses. The significant times for getting the results are the following:

Minimum Retrieval Time: 1ms.

Maximum Retrieval Time: 60091ms.

Average Retrieval Time: 892ms

The figure 6.9 shows the chart of the search time for the 531 entries.

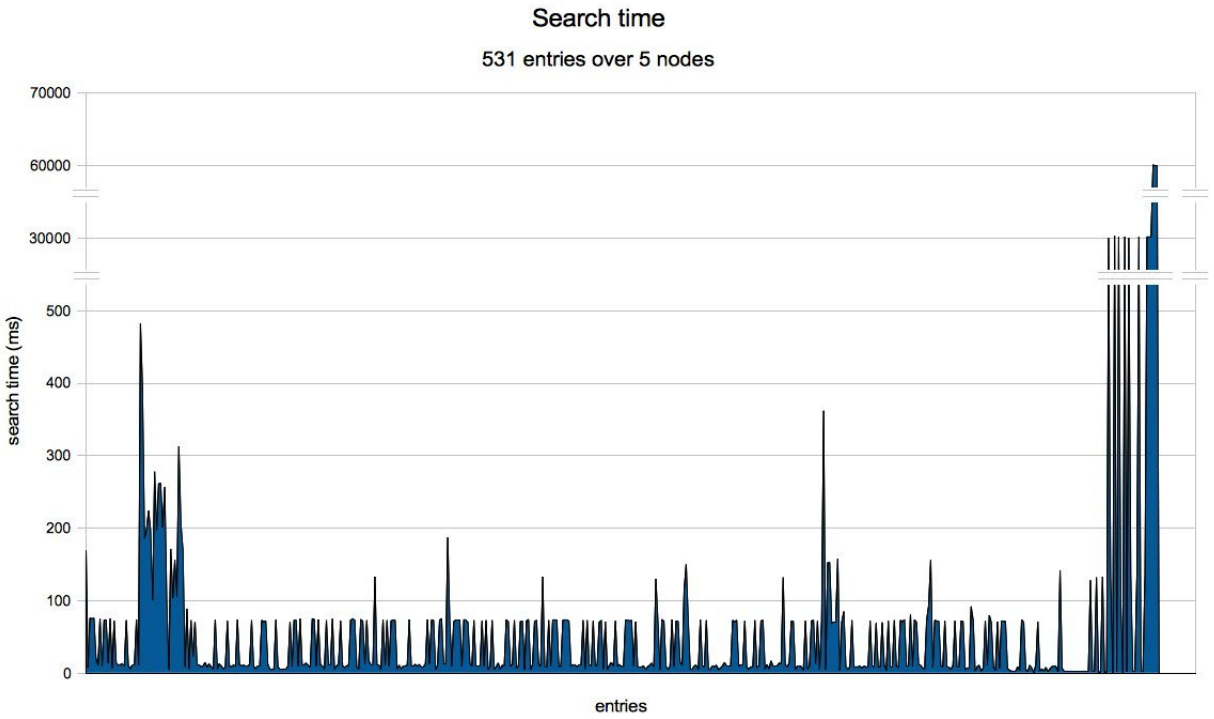


Figure 6.9. Search of 550 entries over 5 nodes on computers connected via ADSL

In some cases, the time needed to get the results is close to 30.000 ms. In other cases, the same time is even closer to 60.000 ms. This behaviour depends on the connection type of the nodes in the Community. The number of cases with a long time of research is small.

6.5.4 A Community of Multiple Nodes on Several Computers

The last test involves 5 computers that are located in the same LAN. Each computer starts 10 nodes. In addition there are two other nodes, one for publication and the other for research. So there is a Community of 52 nodes. The *node_1* runs first creating the new community. The next figure shows the topology of the P2P network.

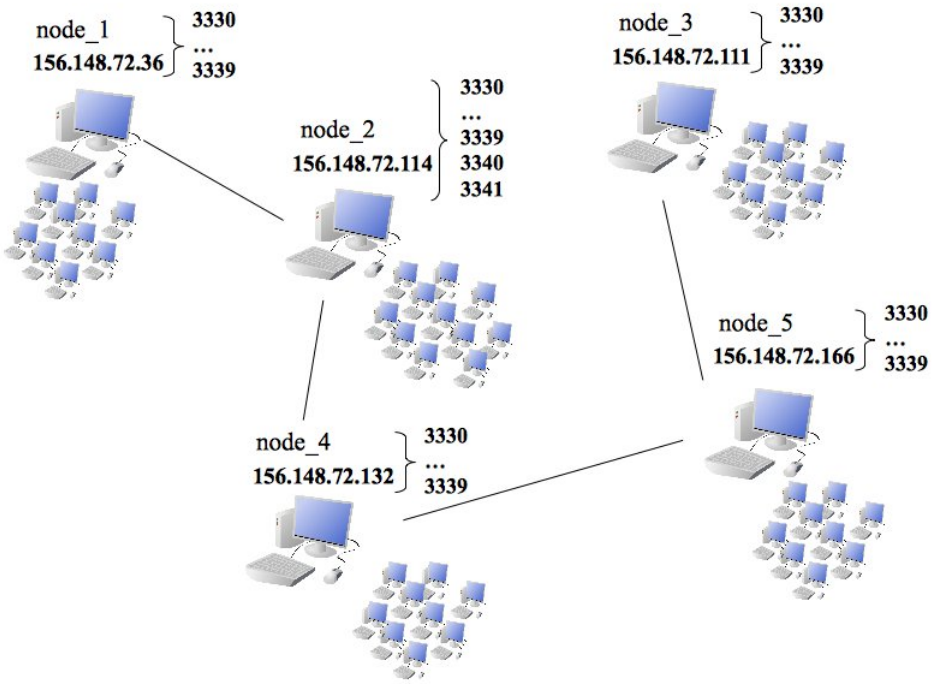


Figure 6.10. Topology of P2P network of 52 nodes on 5 computers

The *node_4*, the one started on the port 3340, runs the publication script. There are no failures during the publication process. The significant times for publishing the entries are the following:

Minimum Publication Time: 7ms.

Maximum Publication Time: 164ms.

Average Publication Time: 31ms.

The figure 6.11 shows the chart of publication times for the 550 entries.

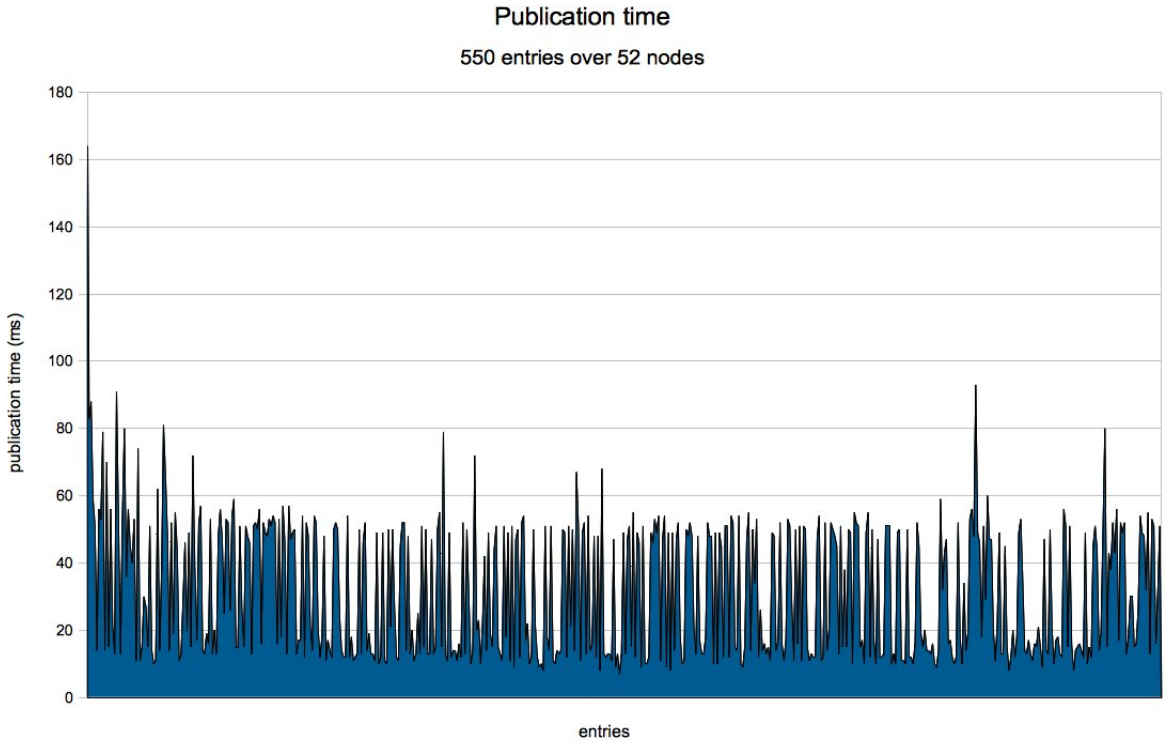


Figure 6.11. Publication times of 550 entries over 52 nodes on 5 computers

The table 6.6 shows the distribution of entries (quota of DHT) among the 52 peers. Each row is related to the 5 different computers located within the LAN. The columns refer to the different peers started on the same node. In addition, the columns *11* and *12* refer to the peers used for the publication of entries and the one used to perform the research.

The sum of all entries is 1.937 (including replicas), so on average there are 37 entries per peer. Considering that we published 550 entries on 52 peers, the distribution is 10 entries/peer. Again, an entry is replicated on three other peers.

Table 6.6. DHT among 52 peers

Peers|*Peers*	1	2	3	4	5	6	7	8	9	10	11	12
node_1	36	61	51	18	44	64	18	28	44	51		
node_2	43	50	17	15	23	28	80	40	54	16	36	44
node_3	21	23	46	44	13	39	63	30	26	51		
node_4	29	41	22	19	18	33	25	45	32	21		
node_5	42	37	27	75	47	32	36	26	38	71		

The *node_2*, the one started on the port 3341, runs the search script. There are no failures in running the search script. The significant times for getting the results are the following:

Minimum Retrieval Time: 3ms.

Maximum Retrieval Time: 373ms.

Average Retrieval Time: 12ms.

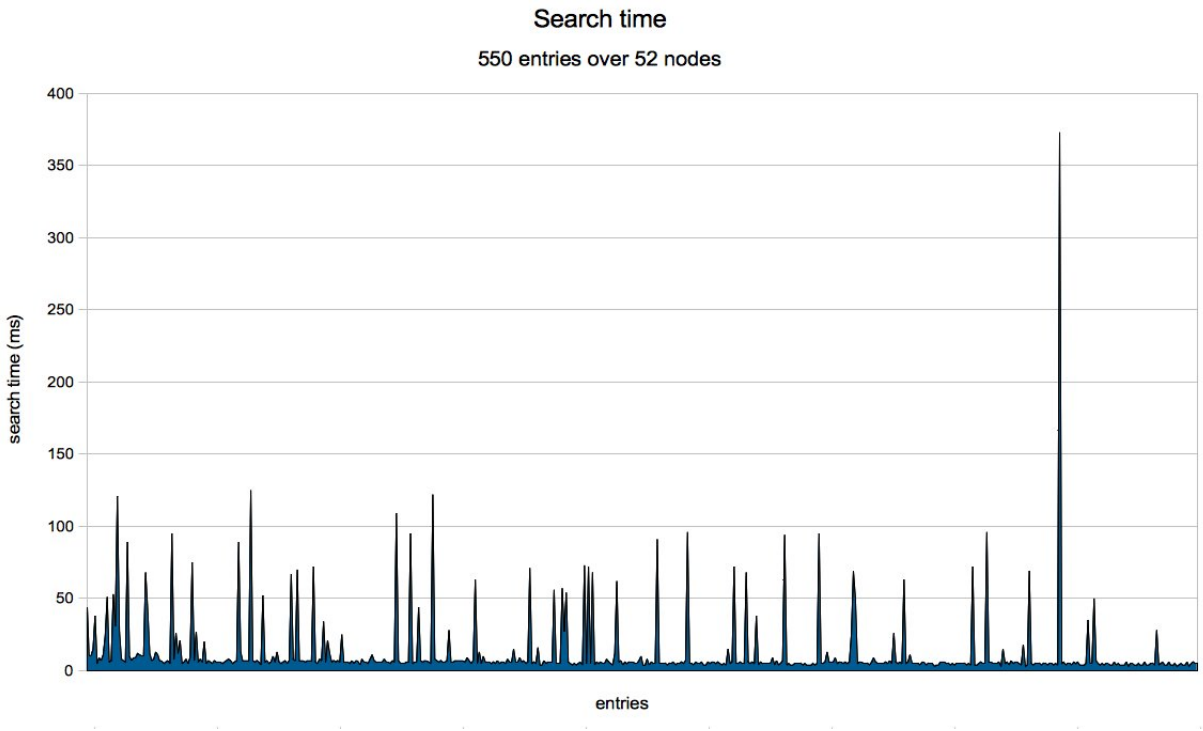


Figure 6.12. Search times of 550 entries over 52 nodes on 5 computers

6.6 Discussion

Our experiments show that the system can effectively support a community of users scattered within a P2P network. We are interested mainly in communities located within private networks such as LANs. At Universities or Companies it is usual to create local sub-networks for internal purposes. We are also interested in Communities constituted by users connected to Internet though ISP, where the issues of firewalling are the main difficulty to overcome. The previous four tests were intended to study these kinds of Communities.

The test set of entries created for these tests is composed of keys of indexing representative of the set of ontologies discussed in this work. The entries contain references to a selection of resources chosen from a local repository. Since there is no need to keep the original name, we decided to simplify the resource file names to the name "Resource". In this way, it was simpler to monitor the publication process following the list of processed resources.

The results of our tests show that the publication time is less than 20 ms, except for the last test. In the last test, the higher number of nodes affect the performance. Even so, the publication time is less than 60 ms. It is an acceptable time and there

are no particular reasons for having a high speed publication process.

The time elapse for searching is, in most cases, less than 50 ms. Sometimes there are higher values, in certain cases close to 60.000ms. These cases are very few. The higher times may depend on the topology of the Community and also on the bandwidth available when the test runs. We can highlight that a long response time is not alarming because the responses of queries are collected asynchronously. Moreover, a query that does not answer in a fixed amount of time can be relaunched, hoping for a more short response time.

When we modified the topology of the Community by disconnecting some nodes, we saw that there was no loss of data within the DHT. This is so because the network is safe-balancing. When a peer dies, the network keeps the same number of replicas of the same entry of the DHT.

Chapter 7

Conclusions

In this chapter we summarize our work, and present an overview of possible future research directions.

7.1 Contributions

In this work we have explored a solution for supporting communities of users who have cultural goals in a specific domain. Community members are interested in managing resources. Initially, the resources are privatly owned by each member. Then, users may agree to share some of their resources with other community members. We have created a system of resource management where each member of the community owns a memory that has a private part, *Personal Memory*, containing personal resources and a public part, *Shared Memory*, containing the documents that have been shared within the community. We consider people who belong to a loose community connected as a network of peer systems.

We based the peer system on a P2P network. To distribute data among thousands or millions of peers involves a huge amount of information and a need for a robust system free of restriction from a central authority. A P2P architecture avoids both physical and semantic bottlenecks that limit information and knowledge exchange. We have chosen to build the P2P network on Pastry, a generic, scalable and efficient substrate for P2P applications. We have used FreePastry, an open-source implementation of Pastry.

We considered it convenient to use the same system of resource management for the resources stored in the two types of memories. For managing the resources, we use *semantic indexing* of the resources. With the semantic index, it is possible to store and later retrieve the resources. The number of queries sent to the system when the resources are searched must be minimized, because they are time consuming. The model of a distributed index is necessarily boolean. In a boolean index,

keys used for retrieving resources must be equal to the keys used for publishing. However, people should be able to find a resource with other characteristics than those used for publishing. The semantic index is composed of entries, pairs of data (*key*, *value*). The *key* of indexing is based on the semantic description that a user gives for describing the resource. The same model of *keys* is used when the resources are searched. The *value* is the URL of the resource. The keys are created using domain ontologies. They are written in OWL and the keys are written in a language based on RDF.

We have defined several cases of indexing. In each case we tackled the process of description and the creation of the keys of indexing. In some cases, it was necessary to perform slight adaptations of the process of creation of the keys. The cases of indexing are based on the possible queries the system can answer. We have based this aspect of the research on two possible kinds of queries, namely *Content Query Type* and *Resource Query Type* that concern the content of a resource and the nature of the resource itself, respectively.

For creating a key, we distinguished between publication and retrieval contexts. The proposed solution foresees during the publication of a resource, different reasonable retrieval situations and different queries to which the resource should respond positively. We used reasoning based on the ontologies involved in the semantic description of a resource. Publication and retrieval contexts are different but may lead to the same resources. Considering the shared memories, the number of queries that are launched through the network are minimized in order to reduce the access time to the resources.

Initially, the user conceives the query in "natural language" and has to rephrase it according to the structure of ontologies. The system considers the input of the user and applies a mechanism based on the available ontologies for helping the user to reformulate the query. The resources that are properly described are then published in the Shared Memory (within the P2P network), or archived in the Personal Memory.

The keys of indexing are created choosing concepts, individuals and relations from the selected ontologies. Depending on the resulting keys, we proposed *algorithms for publishing and searching* resources. We defined *indexing patterns* as the generalization of the cases of indexing that correspond to a path within an ontology and lead to the creation of the keys of indexing. The indexing patterns define a sequence of steps during which the user interacts only with the relevant parts of the ontology. The irrelevant parts are hidden. Selections at each step are inputs to the next step. The indexing patterns are used for presenting the ontologies to users in a friendly and easy-to-use way. We created a mechanism that is able to guide the user during the selection of the important information contained in the ontologies. The user follows a process of indexing, selecting first the ontology to use in order to relate the resources with the elements contained in the ontology. For describing the

different examples presented in this work, we use a test set of ontologies. We also developed a System Ontology for representing the resources of our system and that allowed some cases of indexing.

Users grouped within communities may benefit from the system we developed. Users need tools for accessing the functionality of the community. The system provides a Web *user interface*, equipped with different *tools* presented within the *Semantic Desktop* and integrated within a web application. The web user interface gives a common access to the tools and allows easy communication among all users. The architecture back-end consists of a set of web services for managing the resources and giving access to the P2P network, the Personal Memory and the Shared Memory.

We validated our solution in various experiments. Our experiments show that the system can effeciently support a community of users scattered within a P2P network. We considered various situations where the communities are located within private networks (LANs), or consist of users connected to the Internet through an ISP.

7.2 Future Work

Our work can be improved and extended in several research directions:

- *Exchange with an external system.* An external user is a person interested in discovering resources distributed within the community network and not owner of a peer. Generally, she is using another (preferably semantically based) system for storing resources. She should be allowed to link to a peer of the community and to exchange messages with it.

  An external system can only query our system. It cannot publish resources in the network without being a peer and cannot access to any private memory. In order to access resources of the community, an external system should have to create a semantic description of potential resources based on RDF, communicate it to the selected peer and receive URLs of corresponding resources. Starting from this description, our system would have to create a set of keys and launch corresponding requests to the network.

  However some preconditions are necessary for successful results. The external system must be aware of the ontologies used by the community in order to have adapted results. A specific request for exporting these ontologies could be implemented.

  Publications by external systems are not really usefull because it is necessary to face the issue of delivering a copy of the resource.

- *Multilingual issues.* Many of the elements of ontologies and knowledge bases

are displayed in user interfaces using one of the values of *rdfs:label*. The following example shows the *Document* concept of the System Ontology:

```
<owl:Class rdf:ID="Document">
   <rdfs:subClassOf rdf:resource="owl:Thing"/>
   <rdfs:label xml:lang="en">Document</rdfs:label>
   <rdfs:label xml:lang="fr">Document</rdfs:label>
   <rdfs:label xml:lang="it">Documento</rdfs:label>
   <rdfs:comment xml:lang="en">
     The type of a resource used when describing its content.
   </rdfs:comment>
</owl:Class>
```

There is no problem with adapting tools in the user interface to a particular language as long as ontologies used for indexing are really multilingual.

The issue concerns resources indexed on keywords or indexed on virtual individuals because the user has to add at least one string in order to describe this individual. In case of keywords for expressing natural language text, RDF uses literal nodes. Carrol and Phillips [22] show that literal nodes are either plain literals or typed literals. A plain literal is simply a string and does not contain any type indication. It is not generally appropriate for expressing one thing in different languages (except some specific acronyms). Using additional markup for supporting multilingual issues, the approach we should take is to ask the user to insert different strings for each language intended to be supported and to embed within the description key the literal and the language tag.

- *Evaluation.* An evaluation of the system by a community of users is yet to be performed. A set of experiments should prove that the system can really support the sharing of resources. Collected results might show several aspects of the life of the community, like the use of the wiki and the diffusion of notes. Some comments would be useful for improving the user interface.

- *Advanced navigation system for ontologies.* The user interface could be enhanced with a richer navigation system for ontologies. A graphical system of navigation that allows both exploration of the content of the ontologies through appropriate visualisations and the indexing of resources could be helpful. This would allow one to better organize the visual composition of represented data. A 3-dimensional view would be more efficient for browsing as it involves operations such as zooming, rotating, and translating. The use of different colours allows to add insights on the representation.

- *User profiling.* Creating the profiles of community users would allow to improve the quality of some requests and also their automation. Agents inserted in peer software could warn the user of new publications by regularly requesting the network with preferred and bookmarked queries. Users might be profiled with respect to the tools they use more frequently, the kinds of ontologies they prefer, the kinds of queries they submit, the kinds of patterns they use more frequently, etc. As a result, it would be possible to discover ontologies that the system is missing or ontologies that are incomplete.

Bibliography

[1] Renzo Angles and Claudio Gutierrez. The expressive power of SPARQL. Technical Report TR/DCC-2008-5, Department of Computer Science, Universidad de Chile, 2008.

[2] Ali Arsanjani and Abdul Allam. Service-Oriented Modeling and Architecture for Realization of an SOA. In *SCC '06: Proceedings of the IEEE International Conference on Services Computing*, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Franz Baader. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, September 2007.

[4] Franz Baader, Hans-Jürgen Bürckert, Jochen Heinsohn, Bernhard Hollunder, Jürgen Müller, Bernhard Nebel, Werner Nutt, and Hans-Jürgen Profitlich. Terminological Knowledge Representation: A Proposal for a Terminological Logic. In *Description Logics*, pages 120–128, 1991.

[5] Franz Baader and Ulrike Sattler. Tableau algorithms for description logics. *Studia Logica: An International Journal for Symbolic Logic*, 2001.

[6] Alan Beaulieu. *Learning SQL*. O'Reilly Media, Inc., 2005.

[7] Sean Bechhofer, Ian Horrocks, Carole Goble, and Robert Stevens. Oiled: a reason-able ontology editor for the semantic web. In *Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence*, pages 396–408. Springer-Verlag, 2001.

[8] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. Mcguinness, Peter F. Patel-Schneider, and Lynn A. Stein. OWL Web Ontology Language Reference. W3C Recommendation `http://www.w3.org/TR/owl-ref/`, February 2004.

[9] Dave Beckett. RDF/XML Syntax Specification (Revised). W3C Recommendation `http://www.w3.org/TR/rdf-syntax-grammar/`, February 2004.

[10] Tim Berners-Lee. Semantic web roadmap. http://www.w3.org/DesignIssues/Semantic.html, 1998.

[11] Tim Berners-Lee. Linked data. *W3C Design Issues*, 2006.

[12] Tim Berners-Lee. Notation 3: A readable language for data on the web. Available online at http://www.w3.org/DesignIssues/Notation3.html, March

2006.

[13] Tim Berners-Lee and Mark Fischetti. *Weaving the Web : The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*. Harper San Francisco, September 1999.

[14] Romaric Besançon, Martin Rajman, and Jean-Cédric Chappelier. Textual similarities based on a distributional approach. In *DEXA Workshop*, pages 180–184, 1999.

[15] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes. W3C Recommendation `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/`, May 2001.

[16] Alexander Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori Alperin Resnick. CLASSIC: a structural data model for objects. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 58–67, New York, NY, USA, 1989. ACM.

[17] Alessio Bosca, Dario Bonino, and Paolo Pellegrino. P.: Ontosphere: more than a 3d ontology visualization tool. In *In: SWAP 2005, the 2nd Italian Semantic Web Workshop. CEUR Workshop Proceedings. (2005*, 2005.

[18] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation `http://www.w3.org/TR/rdf-schema/`, February 2004.

[19] Dan Brickley and Libby Miller. FOAF Vocabulary Specification 0.98. `http://xmlns.com/foaf/spec/`, August 2010.

[20] Bill Burke. *RESTful Java with Jax-RS*. O'Reilly Media, Inc., 1st edition, 2009.

[21] Vannevar Bush. As we may think. *The Atlantic Monthly*, pages 101–108, July 1945.

[22] Jeremy J. Carroll and Addison Phillips. Multilingual rdf and owl. In *In European Semantic Web Conference*, pages 108–122, 2005.

[23] Doina A. Cernea, Esther Del Moral, and Jose E. Labra Gayo. SOAF: Semantic Indexing System Based on Collaborative Tagging. *Interdisciplinary Journal of E-Learning and Learning Objects*, 4:137–149, 2008.

[24] Yatin Chawathe, Sriram Ramabhadran, Sylvia Ratnasamy, Anthony LaMarca, Scott Shenker, and Joseph Hellerstein. A case study in building layered dht applications. *SIGCOMM Comput. Commun. Rev.*, 35:97–108, August 2005.

[25] Yatin Chawathe, Sriram Ramabhadran, Sylvia Ratnasamy, Anthony LaMarca, Scott Shenker, and Joseph Hellerstein. A case study in building layered dht applications. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 97–108, New York, NY, USA, 2005. ACM.

[26] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture*

Notes in Computer Science, 2009:46–??, 2001.

[27] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using p-trees. In Sihem Amer-Yahia and Luis Gravano, editors, *WebDB, Proceedings of the Seventh International Workshop on the Web and Databases*, pages 25–30, 2004.

[28] Richard Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170, HP Laboratories Bristol, September 2005.

[29] Ian Jacobs Dave Raggett, Arnaud Le Hors. Html 4.01 specification. http://www.w3.org/TR/html401/, 1999.

[30] John Davies, Dieter Fensel, and Frank van Harmelen. *Towards the Semantic Web: Ontology-Driven Knowledge Management*. John Wiley & Sons, January 2003.

[31] Stefan Decker and Martin Frank. The social semantic desktop. Technical Report DERI-TR-2004-05-02, DERI Galway, Galway, Ireland, May 2004.

[32] Peter Druschel and Antony Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *In HotOS VIII*, pages 75–80, 2001.

[33] M. Duerst and M. Suignard. Internationalized resource identifiers (IRIs). RFC 3987, January 2005.

[34] Marc Ehrig, Christoph Tempich, Jeen Broekstra, Frank van Harmelen, Marta Sabou, Ronny Siebes, Steffen Staab, and Heiner Stuckenschmidt. Swap - ontology-based knowledge management with peer-to-peer technology. In *WOW*, 2003.

[35] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[36] Ian T. Foster and Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In M. Frans Kaashoek and Ion Stoica, editors, *IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 2003.

[37] Valentina Presutti Aldo Gangemi. Ontology design patterns. In Rudi Studer Steffen Staab, editor, *Handbook of Ontologies*, International Handbooks on Information Systems. Springer, 2nd edition, 2009.

[38] O. Ghebghoub, M.-H. Abel, C. Moulin, and A. Leblanc. A lom ontology put into practice. In *Second International Conference on Web and Information Technologies, ICWIT 2009*, Kerkennah Island Sfax, Tunisia, June 12-14 2009.

[39] O. Ghebghoub, M.-H. Abel, C. Moulin, and A. Leblanc. A lom ontology put into practice. In *Second International Conference on Web and Information Technologies, ICWIT 2009*, Kerkennah Island Sfax, Tunisia, June 12-14 2009.

[40] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, June 1993.

[41] Thomas R. Gruber. Toward principles for the design of ontologies used for

knowledge sharing. *International Journal of Human-Computer Studies*, 43(5-6):907–928, 1995.

[42] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *Int. J. Hum.-Comput. Stud.*, 43:907–928, December 1995.

[43] Marc Hadley and Paul Sandoz. Jax-rs: Java api for restful web services. Available online at http://jcp.org/en/jsr/detail?id=311, September 2008.

[44] Andreas Harth. Seco: Mediation services for semantic web data. *IEEE Intelligent Systems*, 19:66–71, 2004.

[45] Andreas Harth and Stefan Decker. Optimized index structures for querying rdf from the web. In *LA-WEB '05: Proceedings of the Third Latin American Web Congress*, pages 71–80, Washington, DC, USA, 2005. IEEE Computer Society.

[46] Erik Hatcher, Otis Gospodnetic, and Mike McCandless. *Lucene in Action*. Manning, 2nd revised edition. edition, 8 2010.

[47] Patrick Hayes. Rdf semantics. W3C Recommendation `http://www.w3.org/TR/rdf-mt/`, February 2004.

[48] Pascal Hitzler, Markus Krtzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language Primer. W3C Recommendation, World Wide Web Consortium, October 2009.

[49] Matthew Horridge, Nick Drummond, John Goodwin, Alan Rector, and Hai H Wang. The manchester owl syntax. In *In Proc. of the 2006 OWL Experiences and Directions Workshop (OWL-ED2006*, 2006.

[50] Ian Horrocks, Dieter Fensel, Jeen Broekstra, Stefan Decker, Michael Erdmann, Carole Goble, Frank van Harmelen, Michel Klein, Steffen Staab, Rudi Studer, and Enrico Motta. The Ontology Inference Layer OIL. Technical report, Vrije Universiteit Amsterdam, Faculty of Sciences., 2000.

[51] Elena Paslaru Bontas Jing Mei. Reasoning paradigms for ow ontologies. Technical Report B-04-12, Department of Information Science, Freie Universitat Berlin, 2004.

[52] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, Bernardo Cuenca Grau, and James Hendler. Swoop: A web ontology editing browser. *Journal of Web Semantics*, 4:2005, 2005.

[53] Zoi Kaoudi, Manolis Koubarakis, Kostis Kyzirakos, Iris Miliaraki, Matoula Magiridou, and Antonios Papadakis-Pesaresi. Atlas: Storing, updating and querying rdf(s) data on top of dhts. *Web Semant.*, 8:271–277, November 2010.

[54] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. Owlim - a pragmatic semantic repository for owl. In *WISE Workshops*, pages 182–192, 2005.

[55] T. Klinberg and R. Manfredi. Gnutella protocol specification. http://rfc-gnutella.sourceforge.net/developer/index.html, June 2002.

[56] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation `http://www.`

w3.org/TR/rdf-concepts/, February 2004.

[57] Nathaniel Leibowitz, Matei Ripeanu, and Adam Wierzbicki. Deconstructing the kazaa network. In *WIAPP '03: Proceedings of the The Third IEEE Workshop on Internet Applications*, page 112, Washington, DC, USA, 2003. IEEE Computer Society.

[58] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *In Proceedings of the 21st annual ACM symposium on Principles of distributed computing*, pages 183–192, 2002.

[59] Frank Manola and Eric Miller. RDF Primer. W3C Recommendation http://www.w3.org/TR/rdf-primer/, February 2004.

[60] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[61] Jeffrey C Mogull. *Representing information about files.* PhD thesis, Stanford University, Stanford, CA, USA, 1986.

[62] Tim D. Moreton, Ian A. Pratt, and Timothy L. Harris. Storage, mutability and naming in pasta. In Enrico Gregori, Ludmila Cherkasova, Gianpaolo Cugola, Fabio Panzieri, and Gian Pietro Picco, editors, *NETWORKING Workshops*, volume 2376 of *Lecture Notes in Computer Science*, pages 215–219. Springer, 2002.

[63] C. Moulin, F. Bettahar, M. Sbodio, J.-P. Barthes, and N. Korda. Adding support to user interaction in egovernment environment. In *4th Atlantic Web Intelligence Conference, AWIC'06*, Beer-Sheva, Israel, 2006.

[64] Claude Moulin, Jean-Paul Barthes, Fathia Bettahar, and Marco Sbodio. Representation of semantics in an e-government platform. In *6th Eastern European eGovernment Days*, Prague, Czech Republic, 2008.

[65] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. Edutella: a p2p networking infrastructure based on rdf. In *WWW*, pages 604–615, 2002.

[66] Natalya F. Noy, Michael Sintek, Stefan Decker, Monica Crubezy, Ray W. Fergerson, and Mark A. Musen. Creating semantic web contents with protege-2000. In *Protg-2000. IEEE Intelligent Systems (2001*, pages 60–71, 2001.

[67] Bijan Parsia and Evren Sirin. Pellet: An owl dl reasoner. In *3rd International Semantic Web Conference (ISWC2004)*, 2004.

[68] A. Passadore, A. Grosso, and A. Boccalatte. An agent-based semantic search engine for scalable enterprise applications. In *Proceedings of the 3rd International Workshop on Ontology, Conceptualization and Epistemology for Information Systems, Software Engineering and Service Science (ONTOSE'09)*, volume 460, pages 82–94, 2009.

[69] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. Owl web ontology language semantics and abstract syntax section 5. rdf-compatible model-theoretic semantics. Technical report, W3C, December 2004.

[70] Peter F. Patel-Schneider and Ian Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax - Section 4. Mapping to RDF Graphs. W3C Recommendation `http://www.w3.org/TR/owl-semantics/mapping.html`, February 2004.

[71] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. pages 30–43. 2006.

[72] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics of SPARQL. Technical Report TR/DCC-2006-17, Department of Computer Science, Universidad de Chile, May 2006.

[73] Woody Pidcock. What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a meta-model? ., January 2003.

[74] Ian Pratt and Jon Crowcroft. Peer-to-peer systems: Architectures and performance. networking 2002 tutorial session,, May 2002.

[75] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation `http://www.w3.org/TR/rdf-sparql-query/`, January 2008.

[76] Gang Qian, Qiang Zhu, Qiang Xue, and Sakti Pramanik. The nd-tree: A dynamic indexing technique for multidimensional non-ordered discrete data spaces. In *VLDB*, pages 620–631, 2003.

[77] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Prefix hash tree: An indexing data structure over distributed hash tables, 2004.

[78] M. Ramos, C.A. Tacla, G. Sato, E. Paraiso, and J.-P.A. Barthès. Dialog construction in a collaborative project management environment. In IEEE, editor, *The 14th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2010)*, volume CD/IEEE Catalog Number CFP10797-ART, 2010.

[79] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, volume 31, pages 161–172, New York, NY, USA, October 2001. ACM.

[80] D. Rodriguez and M.-A. Sicilia. Defining spem 2 process constraints with semantic rules using swrl. In *Proceedings of the Third International Workshop on Ontology, Conceptualization and Epistemology for Information Systems, Software Engineering and Service Science (ONTOSE'09)*, volume 460, pages 95–104, 2009.

[81] Antony Rowstron and Peter Druschel. Storage management and caching in

past, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Oper. Syst. Rev.*, 35(5):188–201, 2001.

[82] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.

[83] Antony I. T. Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, pages 188–201, 2001.

[84] Gerard. Salton. *Automatic Information Organization and Retrieval.* McGraw Hill Text, 1968.

[85] Gerard Salton, Edward A. Fox, and Harry Wu. Extended boolean information retrieval. *Commun. ACM*, 26(11):1022–1036, 1983.

[86] Chatree Sangpachatanaruk and Taieb Znati. Semantic driven hashing (sdh): An ontology-based search scheme for the semantic aware network (sa net). In *P2P '04: Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, pages 270–271, Washington, DC, USA, 2004. IEEE Computer Society.

[87] Leo Sauermann. The gnowsis-using semantic web technologies to build a semantic desktop. Diploma thesis, Technical University of Vienna, 2003.

[88] Leo Sauermann, Ansgar Bernardi, and Andreas Dengel. Overview and outlook on the semantic desktop. In Dennis and Leo Sauermann, editors, *Proceedings of the 1st Workshop on The Semantic Desktop at the ISWC 2005 Conference*, 2005.

[89] Leo Sauermann, Gunnar Aastrand Grimnes, Malte Kiesel, Christiaan Fluit, Heiko Maus, Dominik Heim, Danish Nadeem, Benjamin Horak, and Andreas Dengel. Semantic desktop 2.0: The gnowsis experience. In *The Semantic Web - ISWC 2006*, volume Volume 4273/2006, pages 887–900. Springer Berlin / Heidelberg, 2006.

[90] James G. Schmolze, Bolt Beranek, and Newman Inc. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[91] Michael D. Schroeder, Andrew Birrell, and Roger M. Needham. Experience with grapevine: The growth of a distributed system. *ACM Trans. Comput. Syst.*, 2(1):3–23, 1984.

[92] Fabrizio Sebastiani. Machine learning in automated text categorization. *CoRR*, cs.IR/0110053, 2001.

[93] A. Sheth, C. Bertram, D. Avant, B. Hammond, K. Kochut, and Y. Warke. Semantic content management for enterprises and the web, 2002.

[94] Alessandro Soro and Cristian Lai. Range-capable distributed hash tables. In Ross Purves and Chris Jones, editors, *GIR, Proceedings of the 3rd ACM Workshop On Geographic Information Retrieval*, pages 44–47. Department of Geography, University of Zurich, 2006.

[95] Steffen Staab and Heiner Stuckenschmidt, editors. *Semantic Web and Peer-to-Peer: Decentralized Management and Exchange of Knowledge and Information*. Springer, Berlin, 2006.

[96] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[97] York Sure, Juergen Angele, and Steffen Staab. Ontoedit: Multifaceted inferencing for ontology engineering. *Journal on Data Semantics*, 2800:2003, 2003.

[98] Vanderwal T. Folksonomy: Folksonomy coinage and definition. http://vanderwal.net/folksonomy.html, 2007.

[99] Giovanni Tummarello, Christian Morbidoni, Joackin Petersson, Paolo Puliti, and Francesco Piazza. Rdfgrowth, a p2p annotation exchange algorithm for scalable semantic web applications. In *P2PKM*, 2004.

[100] Mike Uschold. Building ontologies: Towards a unified methodology. In *In 16th Annual Conf. of the British Computer Society Specialist Group on Expert Systems*, pages 16–18, 1996.

[101] Emanuele Della Valle, Andrea Turati, and Alessandro Ghioni. Age: A distributed infrastructure for fostering rdf-based interoperability. In *DAIS*, pages 347–353, 2006.

[102] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. An efficient indexing technique for full text databases. In Li-Yan Yuan, editor, *VLDB*, pages 352–362. Morgan Kaufmann, 1992.