Distributed Search in Semantic Web Service Discovery

by

Joanna Irena Ziembicki

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2006

Abstract

This thesis presents a framework for semantic Web service discovery using descriptive (non-functional) service characteristics in a large-scale, multi-domain setting. The framework uses Web Ontology Language for Services (OWL-S) to design a template for describing non-functional service parameters in a way that facilitates service discovery, and presents a layered scheme for organizing ontologies used in service descriptions. This service description scheme serves as a core for designing the four main functions of a service directory: a template-based user interface, semantic query expansion algorithms, a two-level indexing scheme that combines Bloom filters with a Distributed Hash Table, and a distributed approach for storing service descriptions. The service directory is, in turn, implemented as an extension of the Open Service Discovery Architecture.

The search algorithms presented in this thesis are designed to maximize precision and completeness of service discovery, while the distributed design of the directory allows individual administrative domains to retain a high degree of independence and maintain access control to information about their services.

Acknowledgements

Many thanks to my supervisor, Raouf Boutaba and my readers, Jay Black and Tamer Özsu for their assistance and advice in writing this thesis; Reaz Ahmed for advice on dealing with Bloom filters; Sonia Waharte and Noura Limam for making late nights in the lab bearable; Dave Evans and Rob Warren for their words of wisdom; Carlos Pérez-Delgado for keeping me sane; and all my friends for making grad school so much fun. Finally, I'd like to thank my parents for their love and support, and Grandma Irena for teaching me how to bake in bulk as stress relief.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Consider the following scenario: a busy traveller is searching online for the best way to get from Waterloo, Ontario to Montréal, Québec. She also wants to be able to purchase tickets and book her travel online. Today, the conventional way to accomplish these tasks requires using a keyword-based Web search engine to discover websites of travel booking services, bus lines, train schedules and airlines. Then, the traveller must search each of these sites for the desired connections, compare the options, choose the option for each leg of the trip, and finally, perform online transactions to book the tickets. The drawbacks of the current approach are clear: finding services this way is time-consuming and cumbersome, the web search may not find all the relevant websites while generating numerous useless hits, and each website may come with an entirely different and confusing interface.

In a broader sense, one could envision many types of network-accessible applications, or *services*, which could be discovered by user interaction. Some of them, like online shopping sites, travel-booking agents, or scientific-computing resources, are in common use today. Others, like the plethora of wearable computer-accessible games, "consensual imaging" and communications services described in Vernor Vinge's excellent novella "Fast Times at Fairmont High" [1] seem not far off in the future. Moreover, while some of these applications (e.g. an online currency converter) are an end in themselves, others (e.g. online travel agent) represent real-world services. Both of these categories, however, are similar in that the services involve some sort of interaction over the network. In order to effectively search for these types of services, as illustrated by a few examples of queries in Figure 1.1, at a scale of the current world-wide Web search, we need an Internet-wide architecture that allows user-friendly, efficient service discovery.

A fully-featured world-wide service-discovery architecture that would suppport the sort of search illustrated in the travel-booking example would benefit from the following features:

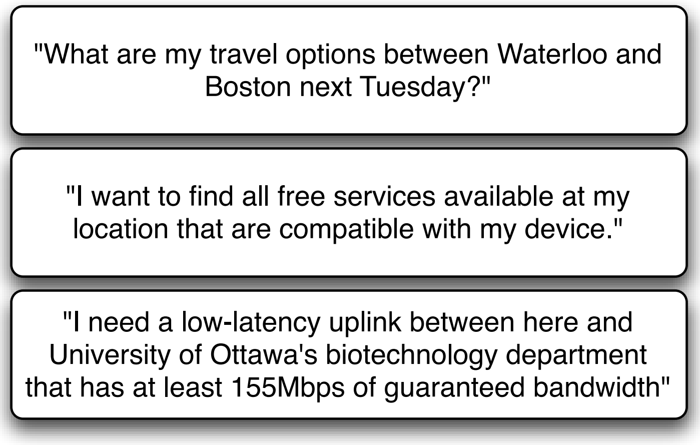

"What are my travel options between Waterloo and Boston next Tuesday?"
"I want to find all free services available at my location that are compatible with my device."
"I need a low-latency uplink between here and University of Ottawa's biotechnology department that has at least 155Mbps of guaranteed bandwidth"

Figure 1.1: Examples of user-initiated qualitative queries

Service description: a well-defined service -description schema would allow users to search by qualitative or *non-functional* characteristics (such as location, price, or network-link speed).

Semantics: effective use of semantics would reduce irrelevant search results and generate as many relevant search results as possible

Unified search: the search architecture should simultaneously search for services from different services providers without previous knowledge of how to access the service providers.

Scalability: the architecture should scale to a large number of users and services (Internet-scale).

There exist quite a few service-discovery systems, such as UPnP[2] or SLP[3], designed for local-area hardware services. However, the Web Services [4] framework is the emerging set of standards for the discovery, description and execution of application-layer services. While Web Service standards provide a rich set of tools for functional-interface description, and for remote execution of services (see Section 2.1), they provide only limited support for the types of non-functional user queries described above. Moreover, the standard Web-Service directory structure does not easily allow universal, scalable cross-domain service discovery. Table 1.1 gives a quick summary of the components of Web-Service standards which support the required features, and of where the standards fail to address the features of the desired service discovery architecture. The goal of this thesis is to “fill in the gaps” in the Web-Service architecture, to allow universal, Internet-wide user-initiated discovery of services.

Requirement	**Supported Functionalities**	**Unsupported Functionalities**
Service description	Well-defined functional description (WSDL [5]), limited non-functional description ("provider", "category", etc. in a UDDI [6] directory)	No well-defined support for flexible non-functional parameters
Semantics	Keyword search on limited parameters	Little support for semantics
Unified search	Each service provider has own directory, directories can be replicated	Usually, requires separate search of each provider; no universal "directory of directories"
Scalability	Supports replication among web-accessible directories, distributing query load	No unified way to distribute service information or index outside of costly replication

Table 1.1: Functionalities supported by Web service standards

1.2 Objective

Our main objective is to devise a framework for discovery of Web services. This framework has two main components: (1) a service description scheme that uses semantic tools to describe non-functional aspects of a service, and (2) a scalable service directory that allows search on these service descriptions across administrative domain boundaries.

1.2.1 Objectives for service description

Before services can be discovered they need to be described in a way that facilitates search. Most current searching algorithms (such as Web search or UDDI search) are based on plaintext keywords. Keyword-based searching is not sufficient for service discovery, since the same words may have different meanings in different contexts; conversely, the same meaning may be described by several different words. In choosing a service description scheme, we need to *move beyond keywords*, and take advantage of semantics in order to direct and expand search.

Another important factor in describing services is the need for flexibility. In widespread, distributed environments, it is infeasible for all services to use an identical description scheme. We need to provide facilities for adding schemas that describe new types of services, to expand existing schemas, and to translate between schemas that describe similar services.

1.2.2 Objectives for service directory

The goal of a service directory is to store service descriptions in such a way that they can be searched effectively by users. A directory whose intended application is Internet-scale service discovery can be expected to store and search a very large amount of information, and is unlikely to consist of a simple database stored on a single host. Therefore, the ultimate goal for the service discovery component of this work is to create a distributed service directory that is able to scale to wide-area, multi-domain applications.

The objective in the design of both the service description scheme and the service directory is not to "reinvent the wheel," but to make the most use possible out of existing tools, expanding them if needed and combining them in novel ways in order to meet the objectives. Because user acceptance is key to the success of any new network architecture, the key objective for this framework is simplicity, of design, of implementation and of deployment.

1.3 Contributions

1.3.1 Contributions in service description

The canonical Web service description consists of a WSDL document describing the interface; the UDDI directory adds its own limited non-functional markup (see Section 2.1). The Web Ontology Language for Services (OWL-S) [7] scheme designed for Web services encapsulates WSDL, and goes a step further in the desired direction, by providing rough-in facilities for non-functional service attributes.

This thesis contributes a structured scheme for describing non-functional service attributes within the OWL-S framework that takes advantage of OWL-S ServiceParameters. This scheme includes a set of restriction on the format of these service parameters and a structured method for organizing the *knowledge base* which contains ontologies used in service description and mappings between these ontologies. This structure allows for user-friendly, template-based creation of service descriptions and queries, and for translating between different service schemas simply by "dropping-in" a file of translation rules. Finally, the service-description scheme allows for semantic search of non-functional service parameters that was not previously feasible in OWL-S.

1.3.2 Contributions in the service directory

The second major contribution of this thesis is a design and implementation of a service directory in a way that takes advantage of the service description scheme. The service directory has four main components.

User interface engine: Takes in user input and turns it into service descriptions or service queries. Chapter 5 discusses in detail the design of the user-interface component.

Semantic search engine: Finds semantic synonyms for service descriptions and queries in order to broaden user search. The algorithms presented in Chapter 6 form the foundation of the semantic search engine.

Indexing engine: Indexes service descriptions to facilitate their future retrieval from the service directory. The indexing engine also processes queries in order to determine which index entries should be accessed to retrieve queried data. An in-depth analysis of this component, which uses Bloom filters [8] and hash functions, is presented in Chapter 7.

Data storage engine: Organizes the physical location of the service description data in the distributed directory using the Chord [9] distributed hash table. The details of the data storage engine are also discussed in Chapter 7.

This partitioning of the service directory is not particularly original, since it emerges from a natural division of tasks. However, each component uses an innovative combination of tools, chosen specifically to best correspond to the features of the service description and to the needs of a multi-domain, wide-area network.

The distributed directory consists of a network of individual *directory nodes*. Each directory node functions as an independent unit, and contains all four service directory components. In contrast to schema-based semantic overlays, each directory node is built around its own core of service-related ontologies, without the need for a common schema for all services.

Unlike many distributed directories (see Chapter 3), the user interface engine eschews complex query languages such as XQL [10], SQL [11] or XQuery [12]. Instead, it uses a simple, user-friendly template-based approach for both service advertisements and service queries. This approach is similar to that of INS/Twine [13] in that it builds queries as a subset of the desired matching service description, but is enhanced with extensible ontology templates and user-defined search semantics. Due to the design of the index and data storage engine, more complex queries can be incorporated easily into the system, by piggybacking them along with simpler queries, but they are not required for the basic functionality of the system.

The semantic search engine takes advantage of the properties of the service description scheme and the knowledge base. It uses these properties together with translation ("glue") ontologies to build *semantic query synonyms*, which allow the user to discover services described using different schemata and different vocabularies. The search is structured enough to allow sophisticated queries, but flexible enough to allow queries on an arbitrary number of attributes.

To allow quick access to distributed service description data, the directory uses a two-level distributed index to answer simple queries quickly and effectively. The local level of the index uses Bloom filters [8] to summarize service descriptions and make them quickly searchable by a subset of their attributes. The global level of the index uses a distributed hash table and category-cumulative per-domain Bloom filters to create a lightweight cross-domain indexing overlay. The indexing approach inventively combines the additive properties of Bloom Filters with the structured peer-to-peer architecture of the Chord [9] distributed hash table, without requiring the hierarchical directory node distribution normally used in Bloom filter-based systems [14] [15] [16], nor the heavy network load typical of Chord-based service directories such as INS/Twine [13].

The main contribution of the final component of the service directory, the data storage engine, is its domain-centric approach. In contrast to the Open Service Directory Architecture [17], on which we base the implementation of this work, the distributed directory architecture centers around a lightweight shared index, but leaves storage and access to service description data to the discretion of the service provider. This approach allows service providers to retain access control of their data, and reduce the amount of (possibly stale) data stored on behalf of other (possibly competing) service providers.

1.4 Case study

Throughout this thesis, we illustrate the service description scheme and the search architecture using a simplified example of a Web service: a provisioning service for optical network links, or *lightpaths*, such as the service being designed by the The User Controlled Lightpath Provisioning (UCLP) project.[1] High-capacity lightpaths are often used in the scientific and research community to transfer large amounts of data between remote physical locations. In many cases, a user may not need a dedicated network link, but only need to use it once in a while, for example, to establish a videoconference between two universities, or to transfer several terabytes of bioinformatics data for processing. It therefore makes sense to use an on-demand Web service to reserve, set up and tear down shared optical links, or concatenate existing links to form longer paths.

Some of the properties of a lightpath include the link capacity and geographical location of the source and destination. Section A.1 of Appendix A shows an example of an optical link service description. Using a simplified lightpath example, we illustrate the rudiments of describing a service, advertising the service in the distributed directory, and using semantics to discover the service.

1.5 Organization

The rest of this work is organized as follows. Chapter 2 provides the necessary background for the technologies and concepts used throughout the remaining chapters. Chapter 3 analyzes related work and compares it to the framework presented in this thesis. Chapter 4 discusses the roles of service descriptions and formalizes a scheme for using an ontology knowledge base describing non-functional service parameters within the OWL-S framework. Chapter 5 uses the organization of a knowledge base introduced in Chapter 4 to

[1]http://www.canarie.ca/canet4/uclp/uclponlab.html

design a template-based approach for the *user interface engine* component of the service directory. The semantic matching algorithms and their role in query expansion are presented in Chapter 6, while Chapter 7 deals with the structure of the index and of the distributed service directory. Chapter 8 gives an outline of the implementation. In conclusion, Chapter 9 sums up the thesis and discusses some future work for extending this framework. For the purpose of illustration, Appendix A gives a few examples of the type of service descriptions and ontologies used in this work.

Chapter 2

Background

The framework presented in this thesis touches on many different aspects of a service directory. As such, it combines tools and concepts from a variety of different areas. This chapter describes the frameworks, technologies and algorithms that have been selected as the basis of the rest of this work: Web service standards, Semantic Web tools and principles of distributed indexing. Its main purpose is to provide the background required to understand the foundations of the rest of this work.

Each of the technologies described in this chapter is used in building one or more components of the service directory presented in this thesis:

Service description: Web service standards 2.1, semantic Web services 2.3

Semantic search: Semantic web tools 2.2

Directory index: Bloom filters, consistent hashing 2.4

Distributing the directory: the Chord distributed hash table 2.4

Implementation framework: Open Service Discovery Architecture

2.1 Web service standards

The term *Web service* can, in general describe any service accessible using Web technologies. More specifically, the World Wide Web Consortium [18] defines a Web service as a software system designed to support interoperable machine-to-machine interaction over a network, and which has an interface described in a machine-processable format (specifically WSDL). Other systems (such as end-user software) interact with the Web service using SOAP [19], which is typically conveyed using XML over HTTP. Additional standards such as the Web Services Choreography Description Language [20] help complete the Web services toolkit, as outlined in Figure 2.1.

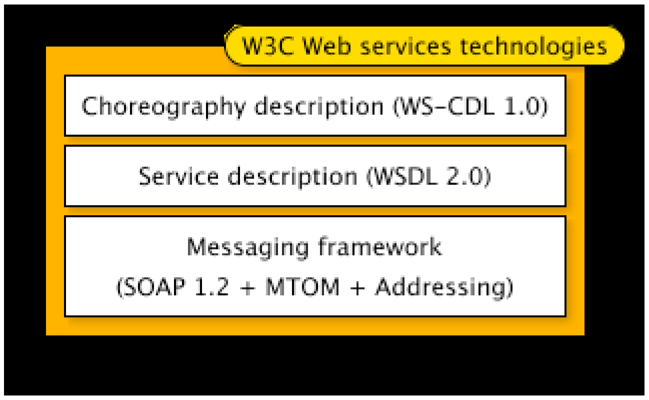

W3C Web services technologies
Choreography description (WS-CDL 1.0)
Service description (WSDL 2.0)
Messaging framework
(SOAP 1.2 + MTOM + Addressing)

Figure 2.1: Web service architecture stack [21]

We call a standalone service that is independent of other services *atomic*. In contrast, a *composed* service is a service that invokes one or more other services. A service composition may involve several services from one domain, or services from different domains. The composition of several services, its message protocols, interfaces, sequences and associated logic is called a *choreography* [18].

2.1.1 Web Service Description Language (WSDL)

The Web Service Definition Language [5] is the standard core language used to describe Web services based on an abstract model of what the service offers. WSDL carries mostly operation-oriented information: it defines the model of interaction with the service, and provides a binding at which the service can be accessed (see Figure 2.2). The messages defined by WSDL are subsequently invoked using SOAP.
A WSDL document contains the following elements [22]:

types	describes the kinds of messages that the service will send and receive
interface	describes the abstract functionality provided by the Web service
binding	describes how to access the service
service	describes where to access the service

The Web Services Description Language (WSDL) has a role and purpose similar to that of Interface Definition Languages (IDLs) in conventional middleware platforms [4]. Conventional IDLs such as CORBA[1] or RMI[2] are tied to a concrete middleware platform, so that they are only concerned with the description of the service interface in terms of service name and signature (input/output parameters). A Web service, in contrast, can

[1]http://www.corba.org/
[2]http://java.sun.com/products/jdk/rmi/index.jsp

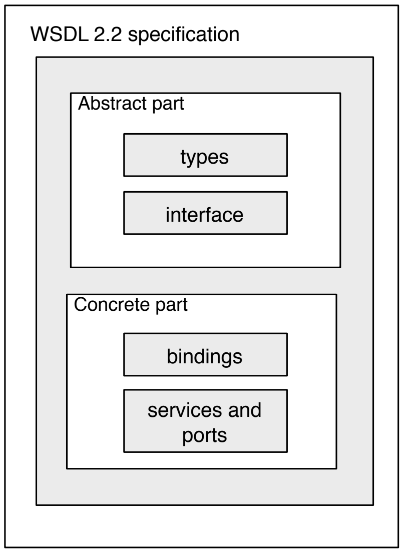


Figure 2.2: WSDL 2.2 specification

be made accessible using different protocols (since SOAP supports bindings to different transport protocols), so it is crucial that such information is provided as part of a WSDL service description. The advantage of WSDL over existing IDLs is that it separates its abstract interfaces from their bindings, allowing for the use of a variety of transport bindings. Therefore, WSDL is more portable and platform-independent.

2.1.2 Universal Description, Discovery and Integration (UDDI)

Structure of UDDI

The Universal Description, Discovery and Integration (UDDI) specification [6] is a framework for describing and discovering Web services. The core of UDDI revolves around the notion of a *business registry*. It provides a simple API that allows providers to register their services by providers, allows end-users to search/browse available services and allows registries to exchange replicas of information. The canonical way to describe the contents of the UDDI business registry is through an analogy with a telephone directory.

White pages	allow discovery of services offered by a given business.
Yellow pages	allow discovery of services based on their category according to a given classification scheme (either standardized or user-defined).
Green pages	describe how a given Web service can be invoked, by pointers to service description documents (typically WSDL).

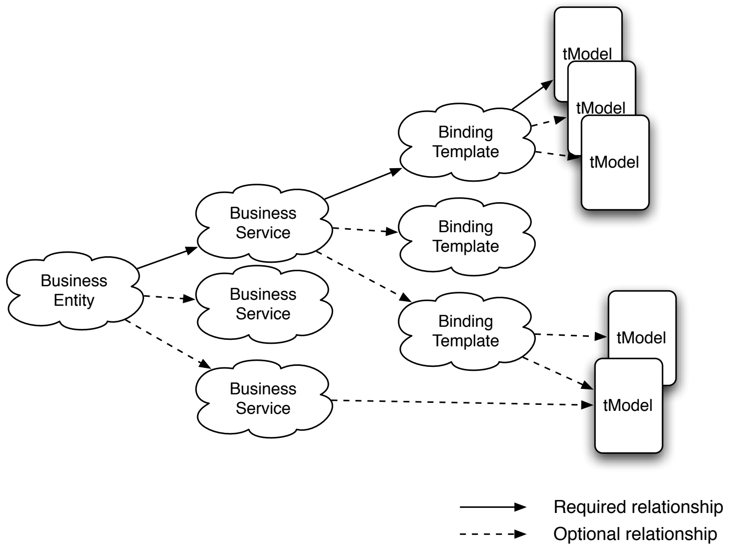


Figure 2.3: One-to-many and many-to-many relationships between UDDI entry components

There are several basic components of the UDDI data model, each of which is associated with its own UUID identifier [6]:

BusinessEntity	(white-pages discovery) describes the organization that provides Web services.
BusinessService	(yellow-pages discovery) describes one or more related Web services offered by a businessEntity.
BindingTemplate	(green-pages discovery) Defines the technical information necessary to use a particular Web service. It contains references to documents (tModels) which describe Web service interface or other service properties.
tModel	(technical models) can contain any kind of specification. Usually, a tModel represents a WSDL service interface document.

Figure 2.3 shows the one-to-many relationships among the different components of the UDDI data model.

The UDDI architecture is centralized, and each business or organization would normally provide a separate (public, private or shared) registry (service directory). Data can be shared between registries through replication.

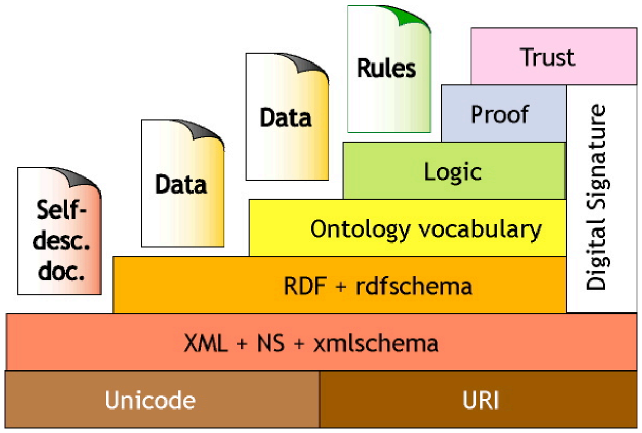


Figure 2.4: Architecture of the Semantic Web

UDDI discovery

UDDI offers a limited set of query facilities to allow end-users to search for services. The model for discovery is that of *browsing*: a user may browse all services provided by a specified business, or all the services corresponding to a specified category. This type of search has two drawbacks: it may return an unmanageable number of results, and requires a high degree of human-user involvement.

Another problem in searching the UDDI registry is that publishers often misinterpret the meaning of the different fields in the UDDI data structures [4]. The missing information, coupled with the highly-structured search parameters required to search a UDDI registry may be detrimental to the completeness of service discovery.

2.2 Semantic Web

The *Semantic Web* provides a common framework that allows *data* to be shared and reused across application, enterprise, and community boundaries. It is a collaborative effort led by W3C with participation from a large number of researchers and industrial partners. It is based on the Resource Description Framework (RDF), which integrates a variety of applications using XML for syntax and URIs [23] for naming, as illustrated in Figure 2.4 [24].

An *ontology* refers to the science of describing entities in the world and how they are related. It is a way to describe the meaning and the relationships of terms. Ontologies specify descriptions for classes (general things in the many domains of interest), the relationships that can exist among things, and properties (or attributes) those things may have.

Ontologies are normally expressed in logic-based languages such as RDF, KIF, OWL

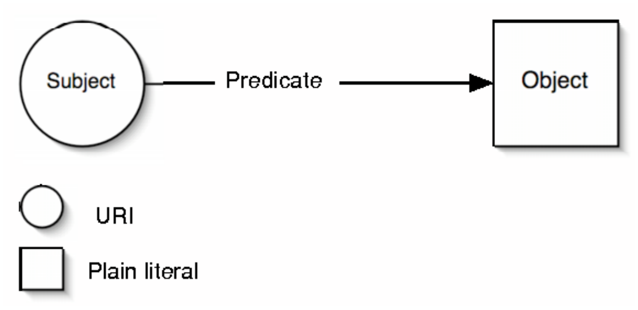


Figure 2.5: RDF (Resource Description Framework) triple structure

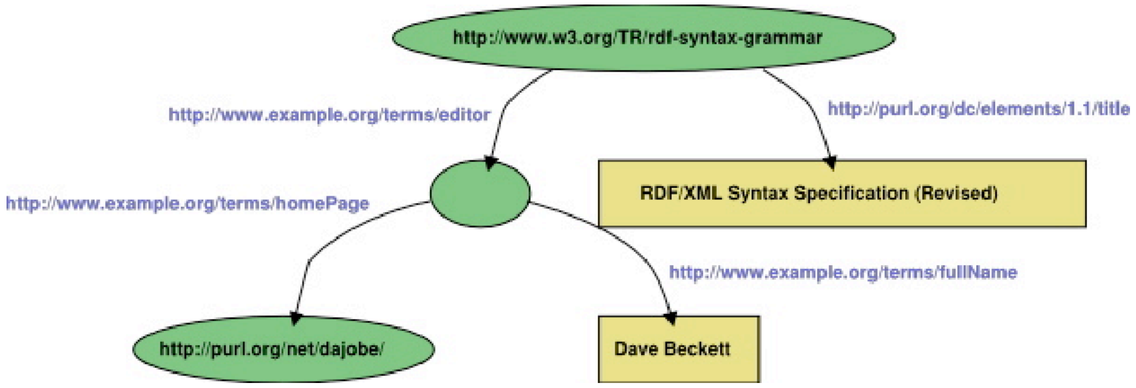

http://www.w3.org/TR/rdf-syntax-grammar
http://www.example.org/terms/editor
http://purl.org/dc/elements/1.1/title
RDF/XML Syntax Specification (Revised)
http://www.example.org/terms/homePage
http://www.example.org/terms/fullName
http://purl.org/net/dajobe/
Dave Beckett

Figure 2.6: RDF example – graph representation [25]

or its predecessor DAML. One of the most basic tools used for writing ontologies is the Resource Description Framework (RDF). RDF a triple-based language created for describing Web resources, but it can be also used for any type of object. It uses URIs and, occasionally, plaintext fields, to identify entities such as objects or individuals, classes of objects and properties of these objects. As seen in Figure 2.5, RDF triples represent a strict $subject \rightarrow^{Predicate} object$ relationship, where the predicate is canonically represented as a directional arrow connecting two nodes. An RDF document can therefore be represented as a directed labeled graph (see Figure 2.6) or translated into an XML representation (see Figure 2.7). RDF Schema (RDFS) is an enhancement to RDF that allows typing, subclassing and treating classes as collections. For example, an RDFS statement allows us to say "every router is a network device" or "if a computer has an IP address then it is a network host."

The more powerful Web Ontology Language (OWL)[3] builds on RDF and RDFS,

[3]From the OWL FAQ [26]
Q. What does the acronym "OWL" stand for?
A. Actually, OWL is not a real acronym. The language started out as the "Web Ontology Language" but

```
<rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
    <ex:editor>
        <rdf:Description>
            <ex:homePage>
                <rdf:Description rdf:about="http://purl.org/net/dajobe/">
                </rdf:Description>
            </ex:homePage>
            <ex:fullName>Dave Beckett</ex:fullName>
        </rdf:Description>
    </ex:editor>
    <dc:title>RDF/XML Syntax Specification (Revised)</dc:title>
</rdf:Description>
```

Figure 2.7: RDF example – XML representation [25]

adding the following capabilities to ontologies:

- Ability to be distributed across many systems,
- Scalability to Web needs,
- Compatibility with Web standards for accessibility and internationalization, and
- Openess and extensiblility.

OWL extends RDFS to allow for the expression of complex relationships between different RDFS classes and of more precise constraints on specific classes and properties [26] (for example, cardinality relations, equality, enumerated classes, characteristics of properties and richer typing of properties). The example in Figure 2.8 illustrates the way OWL can define a class based on properties defined in other classes: here, `TexasThings` are exactly those things located in the Texas geographical region [27].

OWL-based Web ontologies are used in many different applications such as web portals, corporate web site management, bioinformatics and medical communities, and Web services. One of the main efforts towards semantic Web services started out as a project based on DARPA Agent Markup Language (DAML) [28], a predecessor to OWL, and has been migrated to OWL. The following section outlines the effort towards integrating the tools of the Semantic Web with those of Web services.

the Working Group disliked the acronym "WOL." We decided to call it OWL. The Working Group became more comfortable with this decision when one of the members pointed out the following justification for this decision from the noted ontologist A.A. Milne who, in his influential book "Winnie the Pooh" stated of the wise character OWL:

"He could spell his own name WOL, and he could spell Tuesday so that you knew it wasn't Wednesday..."

```
<owl:Class rdf:ID="TexasThings">
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#locatedIn"/>
            <owl:someValuesFrom rdf:resource="#TexasRegion"/>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

Figure 2.8: OWL example – XML representation [27]

2.3 Semantic Web services

While the current Web service description standard, WSDL, describes the message transport mechanism (e.g. SOAP) and the interface used by each service, it does not help in locating Web services on the basis of their capabilities [29]. A step in this direction is the incorporation of Semantic Web concepts and markup in the descriptions of Web services.

A survey [30] of applications of semantic Web services in bioinformatics identifies the following components of Web services.

Service Interfaces , which define access protocols, are published by service providers.

Semantic Description is provided in addition to the interface description. OWL-S (see Section 2.3.1) is the most prominent framework for these descriptions.

A Domain Ontology contains terms describing the key concepts in the subject domain. These terms are used within the semantic descriptions.

Registry/Matchmaker – The matchmaker service searches the semantic service descriptions stored in the registry.

Messaging uses the domain ontology as a *lingua franca* that enables the service consumer to treat data from different providers in a uniform fashion

2.3.1 Web Ontology Language for Services (OWL-S)

Semantic Web services exploit Semantic Web concepts mainly in describing their functional and non-functional properties. The emerging standard for Web service description is Web Ontology Language for Services (OWL-S) which is a set of Web Ontology Language (OWL) ontologies.

OWL-S provides a semantically-based view of Web services with the help of three modules: a *profile* that provides a general description of the Web service, a *process*

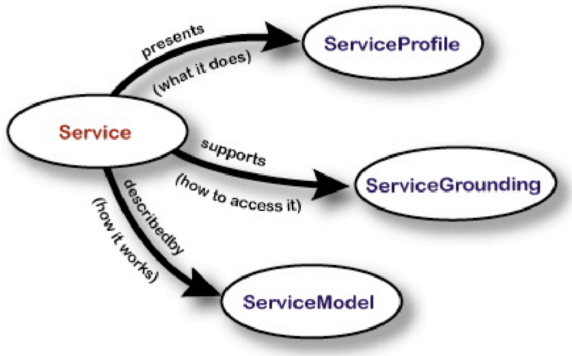


Figure 2.9: Top level of the OWL-S service ontology [7]

model that describes the interaction protocol and tasks of the Web service, and the *grounding* that specifies how the processes in the process model map onto WSDL [31] (see Figure 2.9).

Each of the three modules plays a different role in the service lifetime. The profile of a service is meant to provide a concise description of the service to a registry in a way that is suitable for a service-seeking agent (or matchmaking agent acting on behalf of a service-seeking agent) to determine whether the service meets its needs. After a service has been discovered and selected, the profile is no longer used; henceforth the process model dictates the interaction of the user with the service. The grounding defines the interface of the service at execution time.

From the end-user perspective, the most interesting part of the OWL-S ontology is the *Service Profile* (see Figure 2.10), which outlines what the service provides for prospective clients. It describes a service as a function of three types of information: what organization provides the service, what function is computed by the service, and a host of features that specify the characteristics of the service [7]. The functional description is specified through a set of *Inputs*, *Outputs*, *Preconditions* and *Effects* (IOPEs) involved in the execution of the service. These IOPEs are a subset of those specified in the service model. The description service characteristics include the service category, contact information, a free-form text description, and an unrestricted set of *ServiceParameters*. The latter are not related to the service inputs or outputs, but describe for example, the quality rating, geographic radius, or average response time of a service. Much of the discussion in this work is centered on using these service parameters to provide a better structure to service descriptions.

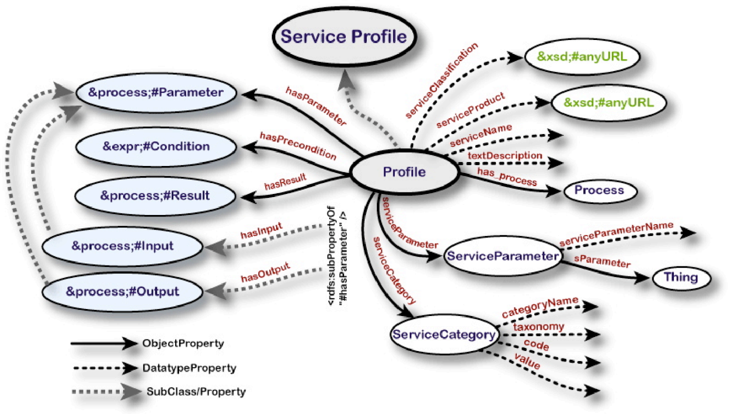


Figure 2.10: Selected classes and properties of the OWL-S service profile [7]

2.4 Distributed indexing

One of the goals of this thesis is to design a service directory that can scale to a large number of advertised services and an arbitrary number of end-users. Clearly, a directory of this size cannot reside on a single web server. There are several levels of distribution: distribution of interface, distribution of the index (first search the index, then search the data, in order to reduce the load on the database) and distribution of the database.

This section presents the foundations for building the distributed index of the directory. First, Bloom filters (see Section 2.4.1) are used to "summarize" service descriptions into a small bit array that can be used to quickly match queries against existing service descriptions. Then, we use the Chord distributed hash table (see Section 2.4.2) to distribute the index over a wide-area peer-to-peer network, using the service category as a quick lookup key. In Section 7.2, these two techniques are used to build the distributed index of the directory.

2.4.1 Bloom filters

The Bloom filter [8], conceived by Burton H. Bloom in 1970, is a space-efficient probabilistic data structure (a binary array) that is used to test whether or not an element is a member of a set. False positives are possible, but false negatives are not. The accompanying algorithm uses multiple hash functions to encode the elements of a set into a single array of bits. The features of Bloom filters, discussed below, make them a great candidate for applications such as distributed databases or cache-membership protocols: in checking whether a data item exists in a specific location, one needs only to receive a small bit array to be checked locally, without unnecessarily querying the remote database. In our system, we use Bloom filters to summarize the contents of service directory nodes.

In Figure 2.11, we illustrate how four hash functions $h_1, \ldots, h_4$ are used to encode a

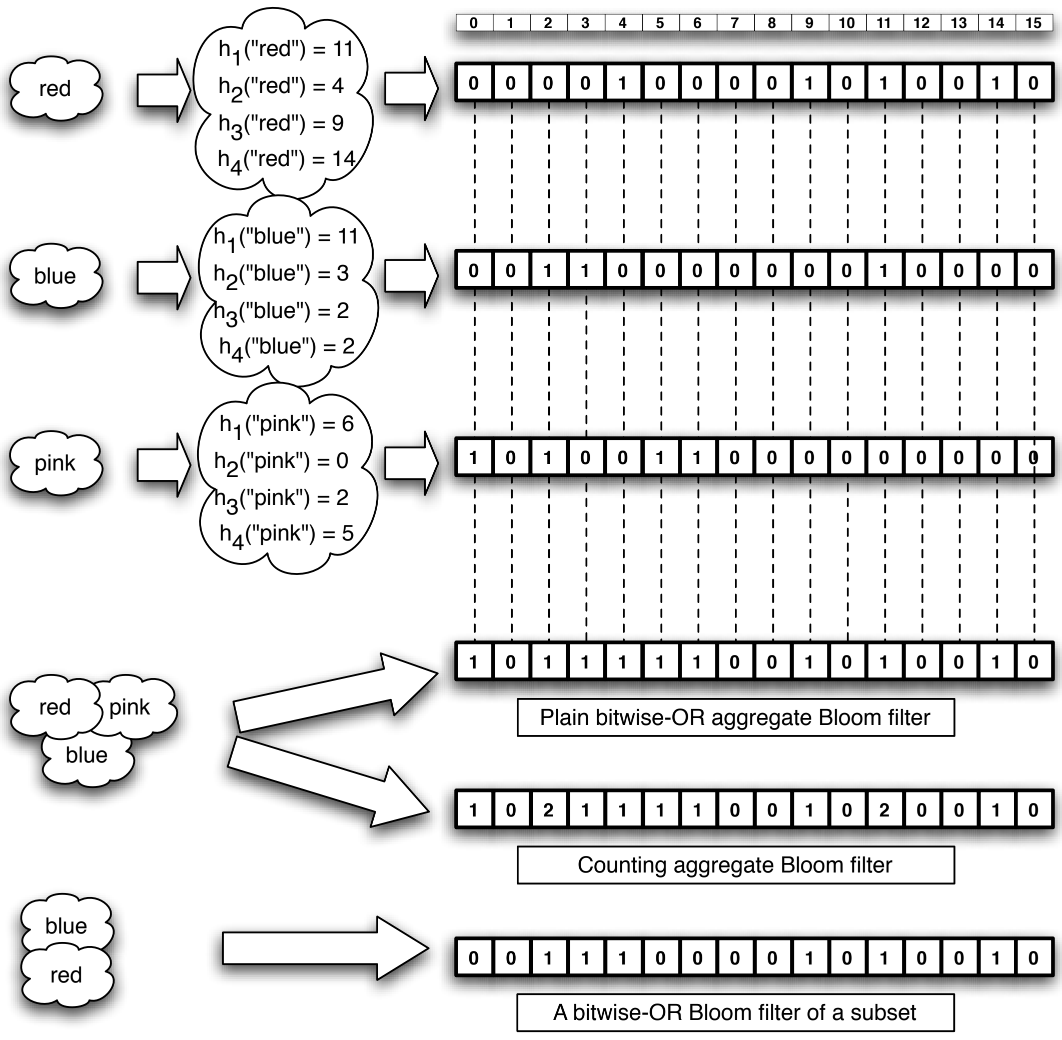


Figure 2.11: An example of a Bloom filter using four hash functions h_1, h_2, h_3 and h_4

small set of strings {"red", "blue" and "pink"} into a Bloom filter B_1. To test whether the set {"red", "blue"} is a subset of the original set, we simply encode it into a second Bloom filter, B_2. If all bits that are set to 1 in B_2 are also set to 1 in B_1, then B_2 is likely to be a subset of B_1. In the figure, we can see that this is the case. On the other hand, note that the Bloom filter for "pink" does not pass the test against B_2, and hence we know that {"pink"} is not a subset of {"red", "blue"}

More formally, let $\mathbb{A} = \{a_1, a_2, \ldots, a_n\}$ be a set of n elements, and let $h_1, h_2, \ldots, h_k$ be a set of hash functions, each with range $\{1, \ldots, m\}$. Given, $\mathbb{H}$, an m-bit array initialized to all zeroes, for each element $a \in \mathbb{A}$, the bits at $h_1(a), h_2(a), \ldots, h_k(a)$ are set to 1. Given an element $a \in \mathbb{A}$ and an m-bit filter $\mathbb{F}$, for the purposes of brevity, let us say that $a \in \mathbb{F}$ when all the bits at $h_1(a), h_2(a), \ldots, h_k(a)$ are set to 1.

Notice that because a Bloom filter is a lossy summary of a set, an insertion into a Bloom filter does not necessarily result in the Bloom filter being changed. If the insertion of a particular item into the filter would result in no change, then testing this item against the original filter results in a false positive. Bloom filters feature a clear tradeoff between m and the probability of a false positive. Observe that after inserting n keys into a table of size m, the probability that a particular bit is still 0 is exactly $\left(1 - \frac{1}{m}\right)^{kn}$. Hence, the probability of a false positive in this situation is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{kn/m}\right)^k.$$

[32]

Another very useful feature of Bloom filters is their additivity. Given two filters F_i and F_j, we define $F_i + F_j$ as the bitwise OR of F_i and F_j. Then given an item a, $a \in F_i$ or $a \in F_j$ implies that $a \in F_i + F_j$. Since the number of false positives is dependent on n/m, consolidating filters in this way decreases the accuracy of the filter. However, the additive property of Bloom filters can be used to create a natural hierarchy of filters, and for checking set containment as well as set membership. Given sets S_i and S_j represented by F_i and F_j, then $S_i \subseteq S_j$ implies $F_i \subseteq F_j$ (i.e. for every bit which is set to 1 in F_i, it is also set to one in F_j).

In order to allow for deletions as well as insertions, a *counting Bloom Filter* can be used to keep a count $c(l)$of how many times a position in a bit array has been set to one (see Figure 2.11 for an example.) When a key a is inserted into or deleted from the filter, $c(h_1(a)), c(h_2(a)), \ldots, c(h_k(a))$ are incremented and decremented accordingly.When a count reaches 0, the corresponding bit is turned off. Note that false negatives can result if too few bits have been allocated to the count value, and a count wraps around to zero.

2.4.2 Distributed hash tables

Distributed Hash Tables (DHTs) are a class of decentralized distributed systems which partition a set of *keys* among participating nodes in a network and can route messages

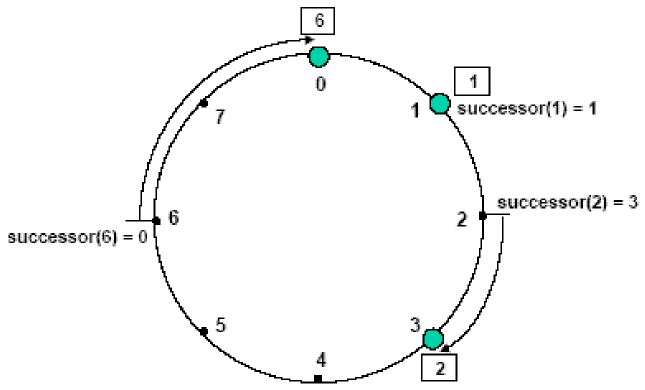


Figure 2.12: Chord DHT ring [9]

efficiently to the unique owner of any given key [33]. They are normally used for efficient content-based routing, where each key is derived from its associated content using one or more hash functions. Proposed applications for DHTs include co-operative web caching, multicast, domain name services and file sharing systems. DHTs were originally developed as a way to provide an alternative to existing peer-to-peer networks: they are more efficient than broadcast-based systems such as Gnutella and guarantee completeness of results, while providing decentralization, scalability and fault tolerance not found in systems such as Napster that require a central co-ordinating server. Examples of DHTs include Chord [9], CAN [34] Pastry [35], Tapestry [36] and Kademlia [37]. The disadvantage of these distributed indexing schemes is that they support only exact search.

One of the earliest Distributed Hash Tables, Chord [9], is used as a basis for the service discovery scheme used in INS/Twine [13]. It is also used in this work, albeit with a different method of generating keys from service descriptions (see Section 7.2). In this case, each Chord node stores $(key, value)$ pairs, where *value* is a service description.

Chord is a DHT first developed in 2001. It treats keys as points on a circle (a "Chord ring") and uses a consistent hash function to map content to keys from 0 to $2^m - 1$ (see Figure 2.12). Given a keyspace of size 2^m, the key 0 is considered adjacent to the key $2^m - 1$. Each of the n nodes in the Chord ring is given an m-bit ID and is responsible for the keys from m to one less than the ID of its *successor* (next node in the ring). In order to allow $O(\log n)$ key insert and key lookup, each node keeps a *finger table* of $m - 1$ pointers to other nodes. For example, a node with ID X will have a finger table pointing to nodes that will map the keys $X + 1$, $X + 2$, $X + 4$, $X + 8$, …, $X + 2^{m-1}$. Given a relatively even distribution of content, the keys and corresponding data values are uniformly distributed among the nodes.

2.5 Open Service Discovery Architecture (OSDA)

The Open Service Discovery Architecture [17] is a middleware for cross-domain service discovery. It aims to serve as a common bridge among the different service discovery schemes used within each domain, such as Service Location Protocol (SLP) [38], Universal Plug and Play (UPnP) [2], Jini [39] or UDDI). OSDA uses an XML-based unified service description scheme (USD, see Figure 2.13) as a common description format, and uses XML-based messaging and SOAP for service description and discovery.

OSDA achieves inter-domain service discovery using a Twine-like [13] approach for USD and Chord-based peer-to-peer overlay (see Figure 2.14) . Each node in the DHT uses an eXist [4] XML database to store USD descriptions. Discovery is done in two steps: first by querying the DHT (which also processes attached XSet-based [40] range queries to further filter the results), then by contacting a *broker* belonging to the target domain in order to retrieve the interface description of the service. The broker may additionally perform intra-domain service discovery to service the request.

The framework presented in this thesis is designed to serve a similar goal as the cross-domain components of OSDA. In fact, it aims to replace the service description component, and enhance the inter-domain peer-to-peer overlay. Unlike the strict format of USD, the service description used in this work is based on a well-known, and more flexible scheme based on web semantics [7]. Searching is enhanced by introducing semantic-matching algorithms. Moreover, the inter-domain overlay used in this work is more lightweight and allows a greater degree of domain autonomicity (see Chapter 7).

[4]http://exist.sourceforge.net/

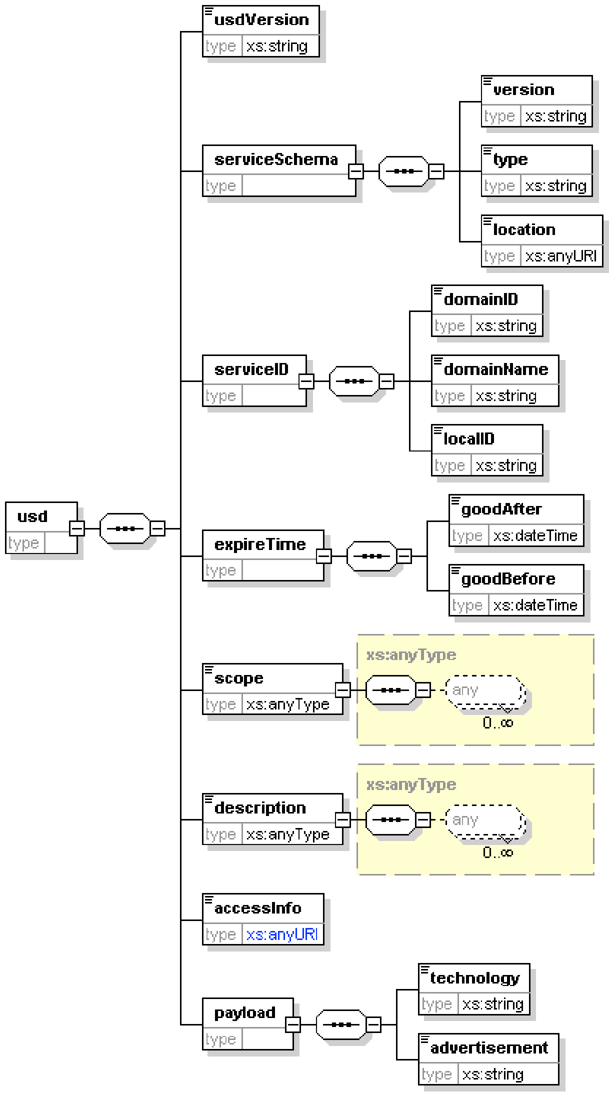


Figure 2.13: Unified Service Description (USD) [17]

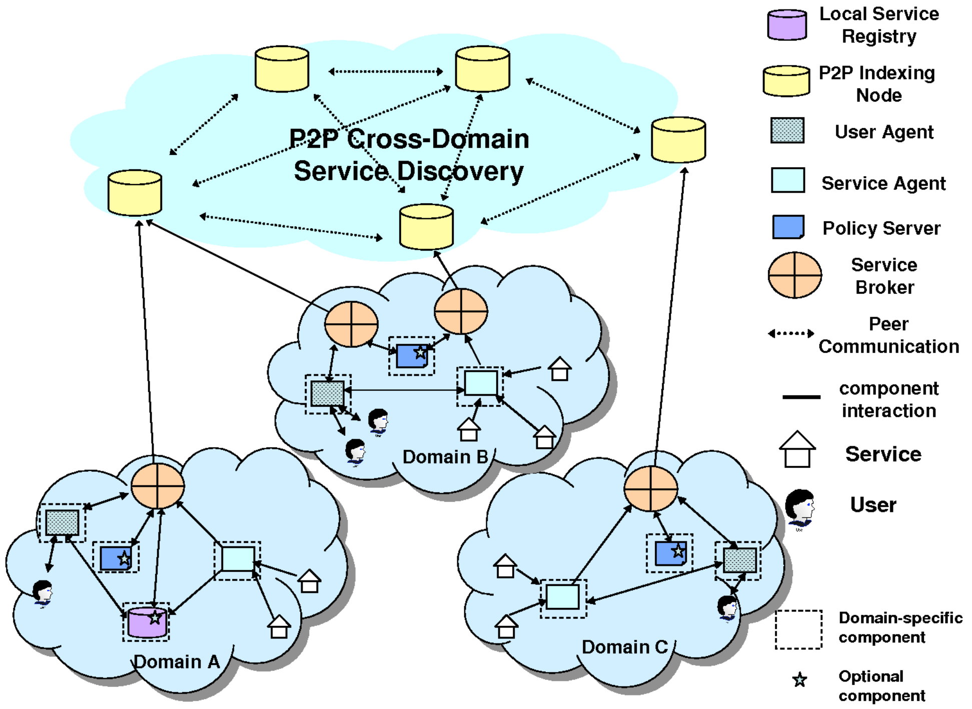


Figure 2.14: OSDA architecture overview [17]

Chapter 3

Related work

In this chapter, we present a number of related works that, unlike the components described in Chapter 2, we did not choose to incorporate into the design of our service directory. Since the design of a service directory spans many different subjects, related works exist in several different areas: indexing techniques for structured and unstructured text, frameworks for using Semantic Web tools in peer-to-peer systems, and distributed approaches in service discovery systems. An exhaustive approach to all of these subjects would be unmanageably long; instead, this chapter presents only the most comprehensive systems similar to the one designed in this thesis, and those tools that would be most relevant to building a similar service directory.

3.1 Plaintext-indexing techniques

Many indexing techniques exist for plaintext search engines. Some of these indexing techniques focus solely on text components such as keywords, phrases and substrings of keywords; others employ natural language techniques to exploit semantics. Common techniques for indexing include inverted files, vector spaces, signature files and bitmaps [41] [42]. The most popular of these approaches, most commonly used in search engines, is an inverted file which links each term in a *lexicon* to its positions in the text database. All these indexing techniques were first developed for a centralized design, but many have been adapted for use in peer-to-peer Distributed Hash Tables using hash functions. These include keyword hashing, partial-keyword search using n-grams [1] [43] [44], and even inverted files [45]. Semantically-enabled indexing techniques, like latent semantic indexing have also been adapted to peer-to-peer storage [46].

Despite their sophistication, natural language indexing techniques are not well suited for OWL-S service descriptions because they do not take advantage of their well-defined structure and their extensive use of exact URIs instead of fuzzy keywords. Natural

[1] n-grams are text fragments of size n

language processing techniques are likely to consume prohibitive amounts of processing power and index space, and may not be precise enough for machine processing. A simpler approach, one suited for structured RDF documents, would be more appropriate.

3.2 Service discovery using hierarchical multi-attribute service descriptions

As with plaintext searching, several distributed indexing techniques have been developed for documents with hierarchical structure, such as XML or attribute-value trees. The two categories of approaches most relevant to this work involve techniques based on Distributed Hash Tables and those based on hierarchical Bloom filters.

3.2.1 INS/Twine

The first approach, INS/Twine [13], is a service-discovery framework that assumes a service description in the form of a hierarchical attribute-value tree. To advertise a service description, INS/Twine extracts *strands*: paths from the root to every node in the tree, and hashes each strand using MD5. All n hashes (where n is the number of nodes in the tree) are then used as keys in a Chord [9] distributed hash table. Service queries are performed by constructing an attribute-value tree of the desired attributes, and performing a DHT lookup using the hash of the longest strand. The query then returns all previously-advertised service descriptions that contain the query as a subtree. Depending on the depth of the tree, the use of strands in indexing can narrow the search to a greater extent than hashing only on service attributes.

INS/Twine is designed for hierarchical service descriptions, and relies on the predictable order of traversal of the hierarchy to index and route its advertisements and queries. While an OWL-S document is in XML format, and therefore hierarchical, its set of service parameters is an unordered flat set. Therefore, an OWL-S based indexing scheme would benefit more from techniques better suited to flat sets of attributes.

3.2.2 MAAN and RDFPeers

RDFPeer [47] is a peer-to-peer repository that stores RDF triples in a Chord-based [9] multi-attribute addressable network (MAAN [48]) by applying globally-known hash functions to each of the components of the triple. A single RDF-triple therefore generates three separate advertisements to the repository. When the advertisement contains a hierarchical RDF document, the document is first converted into a flat list of RDF triples, which are then advertised independently of each other. The system preserves the subject-predicate-object relationships in the RDF triples, but does not make further use of semantics either for routing or for query expansion.

The system allows a rich set of queries, including range queries. It uses a single-attribute-dominated query resolution approach based on choosing the attribute with minimum selectivity as the dominant attribute. This approach has the obvious drawback of presuming that selectivity is known in advance. To allow reasonably efficient range queries, RDFPeers uses a locality-preserving hash on numerical attributes, which similarly requires knowledge of the upper and lower bounds on the attributes, which may be impossible in a generic context.

Because RDFPeers stores each RDF triple as an independent entity, it is better suited for distributing small chunks of information about relationships between entities (e.g. telephone directory) rather than multi-attribute service descriptions that need to be treated as a unit. The system presented in this thesis uses a type of single-attribute-dominated query resolution: the query is dominated by the service category. Our approach is less flexible than that of RDFPeers, but it requires fewer advertisements and is better suited for data with well-defined schemas such as service descriptions.

3.2.3 Bloom-filter-based hierarchical approaches

Bloom-filter-based techniques include Service Discovery Service (SDS) and a hierarchical approach to storing XML service descriptions. SDS [14] is a secure wide-area protocol, which uses XML to encode service descriptions and queries on those descriptions. The service discovery protocol uses a hierarchy of servers that store service descriptions in local XML databases. Each server uses an internal XSet [40] XML search engine to search service descriptions. To accomplish search that spans multiple servers, SDS encodes the service descriptions stored in each server by applying multiple hash functions to various subsets of tags in the service descriptions; this approach is referred to as*Bloom-filtered crossed terminals* . Servers higher up in the hierarchy, consolidate the Bloom filters of servers lower in the hierarchy, allowing query routing within the network of servers. SDS allows searching on multiple attribute-value pairs, but is conceived for a single administrative domain and uses a heavy, broadcast-based approach. The other approach [15] uses the results of breadth-first and/or depth-first traversals of hierarchical XML service description as input to Bloom filters. Its network load is smaller than that of SDS, and the search supports wildcards. Both of these approaches take advantage of the cumulative property of Bloom filters to devise a hierarchical network overlay for storing and propagating the index. In contrast to what is presented in this work, neither of these approaches defines a standard service description scheme.

While both of the above approaches assume a hierarchical organization of service parameters, Bloom-filter indexing is also well suited for indexing flat sets of parameters. However, the natural hierarchical structure of a Bloom-filter index is suitable either for a hierarchical network or a broadcast-based peer-to-peer network, and does not by itself work well in a structured peer-to-peer approach.

3.3 Schema-based peer-to-peer semantic overlays

Another family of approaches to distributed service/resource discovery is that of schema-based semantic overlays on peer-to-peer networks. These overlays group peer-to-peer nodes according to their contents, using simple semantic relationships from a common schema. They are generally designed for applications that assume that all participants in the peer-to-peer network use a small set of common schemas or ontologies. The underlying peer-to-peer infrastructure of the overlays is normally a broadcast-based approach, with unbounded search times and no guarantee of search completeness. The above features make most approaches to schema-based networks better suited for a single-domain application, where "domain" refers either to the administrative domain or to the scope (subject) of the application.

3.3.1 Edutella

Edutella [49] [50] peer-to-peer architecture designed for distributed search of educational content such as lecture slides, books or articles. It proposes describing courseware materials using schemas such as the DCMI (Dublin Core Metadata Initiative)[2], IEEE LOM (Learning Objects Metadata),[3] IMS Global Learning Consortium,[4] and Advanced Distributed Learning Sharable Content Object Reference Model (ADL SCORM).[5] This description metadata is stored in RDF format in distributed repositories.

The Edutella repositories form a super-peer-based peer-to-peer network, arranged in a hypercube topology according to the HyperCup protocol implemented on top of JXTA. The advertisements contain information about the schemata used by the super-peer's group, and are propagated to every super-peer in the network using a non-redundant broadcast. The peers then direct user's queries to the peers whose schema information suggests that they may be able to answer the query. The queries are forwarded wrapped in the RDF-QL-i query language, which can be translated to and from RQL, TRIPLE, SQL, XPath, dbXML, and AmosQL.

In contrast to the scheme presented in this thesis, Edutella is designed for a specific subject domain. While it allows the use of multiple schemas, it only briefly mentions a future mapping service for translating between these schemas, and does not take advantage of RDF semantics. The use of super-peers to aggregate content-provider peers is analogous to the way in which directory nodes aggregate service provider indexes in this work. However, Edutella uses a broadcast-based approach for both advertisements and queries: because there are few guidelines for contents of advertisements, it is possible that each query could reach the majority of peers in the network. This may make Edutella better suitable for a small-scale environment.

[2]http://dublincore.org/
[3]IEEE-SA Standard 1484.12.1
[4]http://www.imsglobal.org/metadata/index.html
[5]http://www.adlnet.org/scorm/index.cfm

3.3.2 The SWAP project and its derivatives

The SWAP project [51] is a complex framework for "combining the peer-to-peer paradigm with Semantic Web technologies," i.e., for storing various types of data using semantics in the description. It proposes a description and rating model that would allow handling heterogeneous ontologies. These ontologies are constructed by a combination of manual and automatic derivation from unstructured data sources such as files, e-mails or bookmarks on a user's personal computer.[6] SWAP uses RDF for both resource description and for annotating individual peers.

SWAP is defined as a loose framework, and does not have specific algorithms for query routing and selection. It suggests a JXTA-based unstructured peer-to-peer architecture, where each peer contains Local Node Repository. These repositories use the Sesame [52] database and can then be searched using the SerQL [53] query language (a select-from-where and construct-from-where filter that uses paths made of RDF triples). The peer-to-peer network uses the concept of peer "expertise": a peer can send advertisements based on the key topics contained in its local repository.

Bibster [54] implements the principles employed by the SWAP project to create a schema-based semantic overlay for sharing bibliographic entries, based on an SWRC ontology[7] and the ACM topic hierarchy[8].

Bibster follows SWAP's unstructured, flooding-based peer-to-peer architecture. Each peer knows about its expertise and finds out about the expertise of neighbouring peers through active advertisement. When a peer receives a SeRQL query, it tries to answer it, or forwards it to other peers whom it judges likelly to be able to answer the query. This likelihood is evaluated by similarity functions between the subject of the query and the previously-advertised expertise topics, using the schema hierarchy and text-similarity methods.

Remindin' [55] extends SWAP with peer-to-peer query-routing capabilities that mimic social metaphors such as observing communication among peers, querying small numbers of peers, and keeping track of meta-information about information stored by other peers. In contrast to the proactive advertisement used in Bibster, Remindin' is a "lazy learning" approach to SWAP, where much of the information used in query routing is based on passive observation. It uses RDFS class relationships to relax queries if necessary, or to suggest similarity between the topics. The peer-to-peer network remains unstructured and based on controlled flooding, but limits the spread of a query by forwarding queries to a maximum of a fixed small number of neighbouring peers.

[6]SWAP Project Deliverable D3.2: Method Design, http://km.aifb.uni-karlsruhe.de/projects/swap/public/Publications/swap-d3.2.pdf

[7]Semantic Web Research Community Ontology, http://ontobroker.semanticweb.org/ontos/swrc.html

[8]http://www.acm.org/class/1998/homepage.html

3.3.3 Concept-based discovery of mobile services

In a recent paper [56], Skouteli et al. propose an architecture for "semantic service discovery in a global computing environment," where data stored on small devices are wrapped and accessed through Web services. Service descriptions are keyword-based, and use the semantic notion of a *concept*, which is a specific property described using a set of keywords.

This concept-based system is ostensibly designed for networks of mobile devices. It uses a hybrid (partially ad-hoc) architecture, where geographical 2-D space is divided into adjacent administrative areas, each managed by a Cell Administration Server (CAS). The CAS manages network addressing, session management and positioning. To mimic the network overlay of CAS nodes, a network of Community Admistration Servers (CoAS) is used to form a semantic overlay of *communities* that provides a semantic index to similar services which may reside anywhere on the network. Each *community* groups services that are semantically-related according to a common ontology (in this case, Google's classification of services in combination with Wordnet [57]). Each CAS stores a "community Bloom filter" for each CoAS in the network, and uses these filters to route advertisements and queries from local devices.

In addition to concept-based keyword queries, this framework also supports queries based on *context* such as location or device-type, as well as continuous queries that notify the user of changes in the result set in a continuous fashion.

3.3.4 H^3

H^3 [58] is a content distribution framework composed of a knowledge infrastructure layer Helios and a communication infrastructure layer Hermes. It proposes to build an overlay network among peers in which each peer maintains a peer ontology describing its knowledge of the network. For query routing, the topology of the overlay networks mirrors the semantic neighbourhood of the peers given by the semantic relationships among the ontologies they own. The work is a preliminary framework of a schema-based overlay, presenting the topics of *knowledge representation*, *matching*, *knowledge distribution* and *semantic routing* only as "future work," without addressing them in detail. Note that the first three of the four topics have been addressed in this thesis in Chapters 4, 6 and 7 respectively.

3.3.5 GloServ

GloServ [59] [60] is a service discovery architecture designed for use in wide area networks. At the time of the first publication, it was based on RDFS, but has since progressed to using the Web Ontology Language as a basis for service description. The *service classification ontology* categorizes the types of services in a hierarchical arrangement and uses OWL's equivalence relations to group related concepts such as "diner" and "deli" into equivalence classes; additionally, every service type is associated with a *service registration ontology*. Advertisements are performed through a service registration template similar to the user interface presented in this thesis. However, queries are constructed by combining ontological and free-text searches using the OWL Query Language (OWL-QL). A *thesaurus ontology* then maps the words in user's query to possible classes and properties within the service classification system, and formulates a query pattern to be forwarded to appropriate GloServ servers.

The hierarchical structure of the service classification ontology is echoed by the proposed arrangement of GloServ servers, which uses a central authority to assign portions of the service classification hierarchy to individual servers. This assignment is stored in a global *location ontology*.

In contrast to this work, GloServ does not propose a structured service description scheme, although the specification briefly mentions the possibility of using OWL-S ontologies as description tools. It also does not adequately address how the user's OWL-QL query is translated into a format useful to the system or whether type equivalence classes are used for query expansion. Finally, the arrangement of GloServ servers and query routing are not fully determined; the cited publications advocate a DNS-like arrangement, while a related technical report [61] proposes the a hybrid hierarchical and DHT-based server arrangement. In the latter, "top-level services" are stored in a hierarchical arrangement of GloServ servers that mirrors the ontology, while servers that store information about the same or equivalent service class are arranged according to a Content-Addressable Network [34] overlay based on that equivalence class. The dimensions of each CAN overlay are associated with the compulsory properties in the service registration ontology of the corresponding type of service. The peer-to-peer structure therefore requires a common ontology and a rigid set of compulsory properties in each service schema. Most notably, GloServ does not consider issues required for a multi-administrative-domain setting: the servers are "public" and information is distributed between them without any notion of ownership. Maintaining a dedicated set of servers for a globally-distributed service directory in this way may hence be practical only for a single-domain wide-area network.

3.4 Semantic matching in Web services

In addition to the distributed service-discovery systems, a number of approaches apply semantic search techniques in a single, centralized service directory. The topic of semantic matching algorithms is a very broad one, with much of the related work dealing with applying semantic search to large databases of plaintext documents. Instead of attempting to summarize a large segment of the field of information retrieval, we restrict the discussion in this section to those approaches which deal directly with Web services or ontology maintenance.

3.4.1 OWL-S based semantic search in UDDI

The OWL-S/UDDI Matchmaker [62] introduces semantic search into the UDDI directory by embedding an OWL-S Profile in a UDDI data structure, and augmenting the UDDI registry with an OWL-S Matchmaking component. The matchmaker is tightly-coupled with the registry, relying on UDDI publish and inquiry functionality for its operations to make a *capability port* that takes over the semantic search. This semantic search is enabled by providing a mapping between the OWL-S Profile and UDDI. Unlike the scheme presented in this thesis, the OWL-S/UDDI Matchmaker searches for services based on inputs and outputs within the IOPEs of the Profile, which is closer to functional matching than to user-oriented non-functional matching presented here. ServiceParameters, in turn, are simply mapped to UDDI TModels, and not used explicitly.

In another departure from the system presented here, this matchmaker uses advertisement-time semantic matching. When an advertisement is published, the ontology concepts in the matchmaker are annotated with the degree of match that they have with the concepts in each published advertisement. The degrees of matching are: *exact*, *subsumption*, *plug-in* (identical to *relaxation* in this work) and *fail*, corresponding roughly to the concepts presented in this work, but evaluated on the level of keywords rather than concept URIs. Indeed, the scheme presented in the paper [62] does not specify explicit OWL-S syntax for annotating these ontologies. The advertisement-time matching has the desirable effect of making the one-time advertisement expensive and subsequent queries inexpensive, but it relies heavily on the centralized nature of the UDDI registry and on the use of a central, static, keyword-base ontology.

A related work [63] uses a simple implementation of an IOPE-based semantic matchmaking algorithm [31] to incorporate semantic search into the UDDI Inquiry API . Semantic markup is accomplished using DAML-S (a precursor to OWL-S).

3.4.2 METEOR-S

The METEOR-S [64] Web Service Annotation Framework is another UDDI-based approach. It is designed to allow semantic search of Web services by semi-automatically matching WSDL concepts to DAML and RDF ontologies using text-based information-retrieval techniques. The matching uses a variety of IR techniques such as synonyms, n-grams and acronym abbreviation. The strength of matches is calculated using a complex scoring formula.

To an even greater extent than the first approach discussed in this section [62], METEOR-S focuses on the functional part (WSDL) of the service description and uses a fuzzy, keyword-based approach, which is not structured enough for the goals of this thesis.

3.4.3 Swoogle

Swoogle [65] is a crawler-based indexing and retrieval system for the semantic web. While not directly related to Web service discovery, it is interesting because of its success and because it highlights differences and similarities between the discovery of semantic web documents and that of semantic Web services. Its crawler-based strategy can be considered a middle ground between advertisement-time and query-time expansion: RDF and OWL documents are crawled and processed at the search engine's leisure, developing a local web of ontologies.

Swoogle automatically categorizes semantic web documents into two layers: semantic web ontologies (SWOs) semantic web databases (SWDBs). Instead of the hyperlink-based set of relationships among documents, Swoogle generates a web of semantic relationships among documents, maintained centrally in a metadata store. Besides following explicit semantic relationships, the indexer uses an n-gram-based word-analysis technique, adding more of an implicit IR flavour to the search engine. User queries, submitted from a simple web-based interface, are evaluated against the internal graph of semantic relationships. The resulting documents are ranked using a custom ranking algorithm.

While Swoogle's loose, Web-like search engine is not the best choice for searching semantic Web service descriptions, it is well worth examining because of its potential in searching web ontologies. A Swoogle-like approach could be used to discover new ontologies and to maintain a dynamically-updated database of these ontologies. Having such a tool available would be very valuable to both the creator of service ontologies, and to the user who searches for services.

In this chapter, we have examined the important literature related to several aspects of the service directory: indexing approaches, use of semantics in distributed search, and practical applications of semantic search in the service registry and the Semantic Web. Studying these approaches has been valuable in determining the set of tools and characteristics that would be most appropriate to the chosen objectives for the design of the service directory, and in determining which techniques are better suitable for other types of applications. Having a good background in related technologies, and having carefully decided which of these technologies would be best for combining in a service directory, we move on to the first main contribution of this thesis: the design of a service-description scheme.

Chapter 4

Service description

Service description is a crucial element of any service-oriented architecture, and plays important roles in service discovery, invocation and composition. An effective service description needs to be expressive enough to help achieve all of the above tasks. In this chapter we take a closer look at service description and its relationship to service discovery, and find that capability-oriented search is poorly-supported in current Web service specifications such as UDDI, WSDL and OWL-S. We propose a structured approach that exploits web semantics to search service directories effectively. To achieve this approach, we build on OWL-S by proposing these additional enhancements:

1. A clear framework for organizing service description ontologies,
2. A well-defined template for service description, and
3. A simple approach that uses the above framework for translating between service description schemas.

It is important to note that this work does not modify the OWL-S standard, rather, it attempts to fill in the gaps left unspecified or glossed-over by OWL-S and its related specifications.

The rest of the chapter is organized as follows. Section 4.1 begins by identifying the purposes of the different components of a service description, and how they are achieved by current Web service standards, highlighting the difference between functional and non-functional components. Based on this discussion, we note that current standards fail in defining service properties and capabilities (non-functional components) in a way that is useful for human-initiated service discovery.

In Section 4.2 we take a closer look at the roles of service description in capability-oriented service discovery, starting out by motivating the need for a more user-friendly way of discovering services. We outline the different types of searches that can be performed on a service directory, and point out the drawbacks of current approaches. We follow with a detailed discussion on the ways that ontologies and semantic markup can enhance service discovery, identifying the OWL-S Profile as a promising though under-specified tool in capability-oriented search.

Section 4.3 defines a hierarchical framework for organizing service description ontologies, and an organized approach for ontology matching and translation. The clean separation of concepts demonstrated in this section serves as a foundation for creating well-structured service descriptions that lend themselves well to indexing and generic approaches to semantic matching.

Finally, Section 4.4 aligns the concepts of OWL and OWL-S ServiceParameters with the ontology framework proposed in Section 4.3. It introduces a scheme that uses OWL-S ServiceParameters to define non-functional service attributes to be used in semantically-enabled, capability-oriented service discovery.

4.1 Purposes of service description

Web service descriptions aim to serve several different purposes. Some of these purposes cater to humans: they describe, in human-readable terms, the attributes and functionalities of the service. Other purposes are meant for machine use: they describe the service interface and facilitate the execution of Web services. In general, the goal of Web service descriptions is to achieve the following three functions [7]: Web service discovery, Web service invocation and Web service composition.

In the rest of this section, we examine elements of service description and their role in achieving the first functionality, while remaining aware of the support for the latter two. In particular, we focus on differentiating between functional service attributes and non-functional service attributes, their roles in service description, and how these roles are achieved in current Web service standards.

4.1.1 Functional vs. non-functional components

Service descriptions are composed of functional and non-functional components. **Functional components** comprise the information required to execute a service: the interface specification, the associated business protocols, the inputs/outputs and preconditions/effects (IOPEs) of the service. The **non-functional components** contain descriptive information that is not immediately relevant to service execution. They include such properties as provider information, geographical scope and cost of a service, as well as abstract capability descriptions such as "provides GPS resolution of 10 metres" or "available in English, Swedish and German."

4.1.2 Service description functionalities of current Web service standards

The standard Web service description that encompasses functional and non-functional parameters has several layers of abstraction: the common base language, the interfaces,

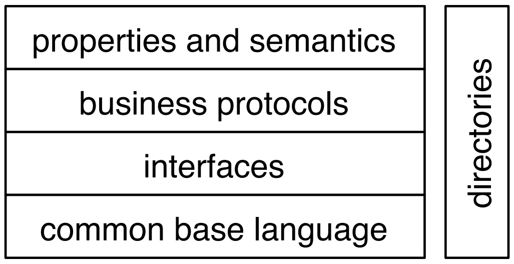


Figure 4.1: Service description and discovery stack [4]

the business protocol and the properties and semantics (see Figure 4.1). We examine these layers from the bottom-up:

Common base language: XML is generally used as the common language for expressing Web service descriptions.

Interfaces: The dominant interface description language used in Web services is WSDL. This layer defines how end users (including other software programs) must interact with the service, and is used primarily for invocation. It can also be helpful in determining composability.

Business protocols: This layer is used for defining business service flows to aid service composition and choreography. Composition and transaction languages include WSCL (Web Service Conversation Language), BPEL (Business Process Execution Language for Web Services) and WSCL (Web Services Choreography Description Language) [20].

Properties and semantics: This layer defines the non-functional properties of a service such as the cost or quality of the service. These properties are an essential tool for service discovery.

The first three components: *common base language*, *interfaces* and *business protocols* can be seen as functional components of service description. The "upper-layer" *properties and semantics* consist of a set of non-functional properties.

In the current set of Web service standards, the functional components that describe interfaces are reasonably-well defined: standards such as SOAP and WSDL are designed to provide descriptions of message transport mechanisms, and of the interface used by each service. However, UDDI does not provide enough flexibility in the components describing properties and semantics: it provides only rudimentary treatment of non-functional components (e.g. category, provider, etc.). Similarly, neither SOAP nor WSDL are of any use for automatic location of Web services on the basis of their capabilities [29].

As in UDDI and WSDL, the majority of OWL-S ontology components (the Service-Model, ServiceGrounding and the IOPEs included in ServiceProfile) focus on functional service attributes. However, as outlined in Section 2.3.1, the ServiceProfile offers primitive facilities for non-functional service description. The few prescribed non-functional components, *serviceName textDescription*, *contactInformation* and *serviceCategory*, offer little flexibility beyond that of UDDI. On the other hand, the under-specified *serviceParameters* can be exploited to provide a richer capability description of the service, using the semantic tools of OWL and RDF.

In addition to being specified only on a basic level, most of the non-functional components of Web service descriptions are intended mainly for human consumption. To allow the automation of service discovery these components need machine-readable semantics. The rest of this chapter focuses on enhancing the service-discovery function of service description using semantic markup of its non-functional components.

4.2 Role of service description in service discovery

One of the most important roles of service description is to be stored in a directory for the purpose of service discovery. Service discovery can be done both at design-time, by browsing the directory and identifying the most relevant services, and at run-time, using dynamic binding techniques [4].

In this work, we concentrate on user-initiated service discovery using non-functional rather than functional components. When looking for a service, a user is unlikely to query the directory using details of the interface or types of inputs. Rather, the user would more likely specify qualitative queries such as in Figure 1.1.

The examples of user-initiated qualitative queries illustrate that Web services represent not only software entities but also ways to interact with real-world occurrences (for example, an online travel-booking service represents a real-world transportation service). The translation of free-form user requests to queries that the system can understand is largely a natural-language problem, and beyond the scope of this work. Instead, we discuss a simpler setup for query generation in Chapter 5. For now, we assume that directory queries have already been specified in an appropriate format. When the more abstract, capability-oriented requirements are satisfied, the search can be narrowed further by functional attributes to determine composability and supported method of invocation. There are several types of searches that can be performed on a service directory:

Keyword matching: textual search is based on a set of keywords, typically used in World Wide Web searches.

Browsing: a set of service descriptions in a directory is browsed manually by a user. This set can be narrowed down by a single attribute such as *company name* or *category.*

Attribute-value matching: service descriptions are matched based on a set of attribute-value pairs (this is how discovery is done, for example, in INS [13] and SLP [38] service discovery systems)

Semantic matching: a combination of keyword and/or attribute-value matching can be used, along with a set of semantic rules.

When it comes to service discovery, UDDI is limited to keyword matching or exact matching of service descriptions, and does not support any kind of semantic processing that would be necessary for flexible matching of service advertisements to service queries. [66]. As mentioned in Section 2, searching a UDDI directory is normally performed by browsing a set of service descriptions narrowed to a given provider or category.

Given a well-known set of categories, and a relatively limited set of services, narrowing a search by category can be sufficient for a first step of discovery. However, in a large-scale directory, searching by a specific category alone can be either too broad or too narrow.

Searching by category can be too narrow when several different classification systems are used, resulting in different category names being used to define similar services. In this case, it is better to search by attributes, or to have rules that relate these categories to each other. Searching by category does not allow, for example, to perform broader types of discovery, such as "Find all services that are available at my current location, and can be invoked by my cell phone." AT the same time, a category-only search can be too broad in that it may return many useless hits of services that match the category but not other desired attributes. Relevant hits can become lost in the noise of non-relevant hits.

4.2.1 Semantic description and service discovery

The goal in devising a successful service description scheme is to make it detailed enough and structured enough that a user does not need to browse all available services to discover a desired service. In order to allow more sophisticated and flexible user-oriented service discovery, we need to strike a balance between the flexibility of ad-hoc keyword-based search used for searching web documents, and the precision of the highly-structured search of UDDI registries.

The impact of semantics on the vocabulary of service descriptions has two aspects, both of which affect the effectiveness of an attribute-based search: the narrowing aspect and a broadening aspect, to offset the "too broad" and "too narrow" aspects of category search from the previous section. The narrowing aspect *grounds the vocabulary in a common ontology*, providing context otherwise absent from keywords. For example, the attribute "colour" has a different meanings in the context of a car and in the context of a printer. This aspect narrows attribute-based search, by eliminating out-of-context attributes. In turn, the broadening aspect *provides a relationship to other vocabulary* *concepts*. Applying ontology concepts to service descriptions allows us, for example, to define “colour” as a synonym of “color” and “couleur,” and allows us to define “sedan” as a subclass of “car.” This aspect lets us broaden attribute-based search by including a specified set of related concepts in the results, by using subsumption, for example.

The use of URIs and XML namespaces provides the tools to achieve the narrowing aspect, since every concept can be uniquely determined by a URI as part of a given workspace. In order to achieve the broadening aspect, we need more sophisticated tools for expressing keyword and concept relationships, such as RDF and OWL. To take full advantage of the expressive power of OWL, while restricting the description to a standard format, we choose to use OWL-S as the basis for the service description.

The OWL-S *serviceParameters* (see Figure 4.2) are an expandable list of properties that may accompany a service description. They are defined as an OWL ObjectProperty (see Figure 4.3) , and can be subclassed to express a variety of different concepts. While traditional service description schemes normally restrict properties to a set of fundamental types (string, integer, enumeration), or rely on plaintext keyword matching, the rich set of OWL relationships can be used to express *serviceParameters* as complex objects with arbitrary sets of their own properties. Both the “attribute” and “value” components of a service parameter can by themselves be concepts in a domain ontology. Together with a well-defined set of ontologies, OWL-S service parameters are a promising tool in defining the non-functional components of service descriptions.

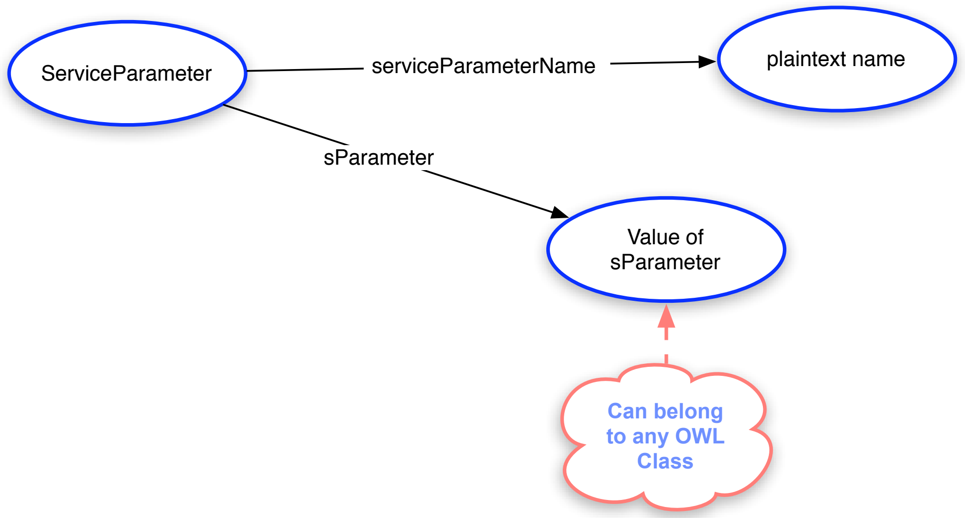


Figure 4.2: Basic structure of OWL-S ServiceParameter

Before continuing the discussion of how OWL-S can be extended to allow effective capability description, we begin by discussing its drawbacks. OWL-S is a fairly immature

```
<owl:ObjectProperty rdf:ID="serviceParameter">
        <rdfs:domain rdf:resource="#Profile"/>
        <rdfs:range rdf:resource="#ServiceParameter"/>
</owl:ObjectProperty>
```

Figure 4.3: An OWL definition of *serviceParameter* in the OWL-S specification [67]

technology that needs further refinement before it can live up to its full potential as a service description model. The latest version of OWL-S (OWL-S 1.1) has already fixed the OWL syntax errors of previous versions, and made OWL-S compliant with the fully-inferrable OWL-DL [66]. Still, even the latest OWL-S specifications suffer from conceptual ambiguity, poor axiomatization (e.g. many relations/concepts take *owl:Thing* as their domain or range), loose design and narrow scope [68].

The general consensus therefore seems to be that OWL-S is under-specified; it contains too much ambiguity, and lacks precision in some of its concepts. In particular, it needs a more structured approach to designing and organizing ontologies. To allow better machine-based processing and inference, a better relationship needs to be constructed between OWL-S and foundational ontologies, so that both the functional and non-functional parts of the description can be described in a more rigorous manner.

These principles align well with the purpose of this work. Clearly, searching and analysis of service descriptions would benefit from a clear relationship of the service parameters to well-defined knowledge domain ontologies. Much work on semantic language analysis has been accomplished in the areas of linguistics and cognitive science; while the complexity of analysis of service descriptions should never approach that of natural language processing, it can certainly benefit from the rigorous definition of concepts (for example of time theory and causality) from foundational ontologies such as DOLCE [69]. However, this thesis concentrates on exploiting generic rules and simple relationships between ontology concepts, leaving complex inference to specialized applications.

4.3 Managing service ontologies

One of the main problems in managing a semantically-enabled service directory is keeping track of ontologies involved in service description. There are several types of these ontologies. The first type of ontology (such as an OWL-S or DAML-S document) defines the service description itself. Another type of ontology defines the *knowledge domain* of the service, and contains the vocabulary used to describe the service presented as a network of concepts and relationships. These ontologies can be combined or spread over several files. This unstructured approach is flexible, but can make it difficult to process ontologies in a generic fashion.

This section describes two of the main contributions of this thesis: a framework for clean separation of service description ontologies and a simple mechanism for ontology translation.

4.3.1 Ontology layers

In describing a service, there is a natural division of information. First, we have a vocabulary of concepts for describing the service. For example, a hotel reservation service would need a vocabulary for geographic locations (cities, countries), currency and credit card types, while a lightpath service would need to describe and relate units of bandwidth, network addresses and supported protocols. We call this layer the *knowledge domain* of the service. At a more general level, each service category is associated with a canonical set of attributes whose values would be taken from the knowledge domain. For example, the stock parameters of a lightpath service would include *source IP address*, *destination IP address*, *link capacity* and *supported QOS classes*. These parameters form the *service category* layer. Finally, the *service description* populates the parameters according to its category with elements from its knowledge domain. We therefore divide service ontologies into three distinct layers:

Knowledge domain ontology: contains vocabulary associated with the service. This layer can be further subdivided into ontologies that describe the relationships between classes of objects, and the object instances themselves.

> **Types:** this ontology contains the classes of items ("country", "province", "city")
>
> **Instances:** this ontology contains the instances of the types ("Canada", "Ontario", "Waterloo Region", "Waterloo")

Category ontology: contains the pre-defined set of attributes associated with a service category. Besides defining attributes for specific service categories, this layer can also define a set of generic attributes that can be associated with a wide variety of services (e.g. geographic scope, expiry time, access-control information).

Service description ontology: is the actual service description document, containing attributes chosen from category ontologies and values chosen from the knowledge domain ontologies.

4.3.2 Mapping between ontologies

There is usually a close bond between the category of a service and its knowledge domain ontology. However, it is possible for several different ontologies to be used to describe the same service category. One of the main problems in managing knowledge domain ontologies is mapping or translation between concepts in different ontologies. For example,

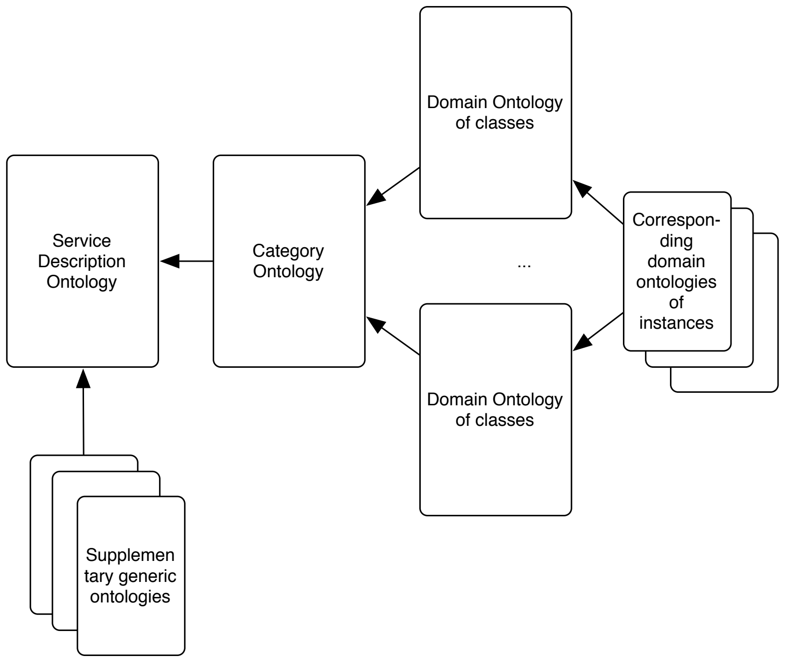

Domain Ontology of classes
Service Description Ontology
Category Ontology
...
Correspon-ding domain ontologies of instances
Domain Ontology of classes
Supplemen tary generic ontologies

Figure 4.4: Ontologies stored in a service directory

given several descriptions of wireless base stations, we want to make sure that "transmission radius = 100 metres" in one ontology represents the same concept as "broadcast range = 100m" in another.

There are several approaches to ontology translation. Broader areas, such as text processing can benefit from probabilistic, AI-based approaches [70]. A more explicit approach is to combine ontologies to form *knowledge spaces* [71]. The higher level of structure inherent in semantic service descriptions as well as the desire to preserve the narrowing aspect of semantic markup (see Section 4.2.1) lend themselves to a very simple approach to creating knowledge spaces: a *glue ontology*.

In our approach, a glue ontology uses semantic tools such as OWL's `equivalentClass`, `subclassOf` or `sameAs` to specify relationships between the vocabularies of two or more different ontologies (as illustrated in Figure 4.5). This mapping can be performed both between the types (classes) of objects, and between object instances. In service directories, the most useful relationships involve equivalence and subsumption: they help determine, for example, whether one service can be used in place of another.

Using the ontology hierarchy designed in Section 4.3.1, we can use glue ontologies at two layers: between knowledge domain ontologies, and between category ontologies (always "gluing" ontologies from the same layer). The latter case can help solve a very important problem in service directories: mapping between two different service descrip-

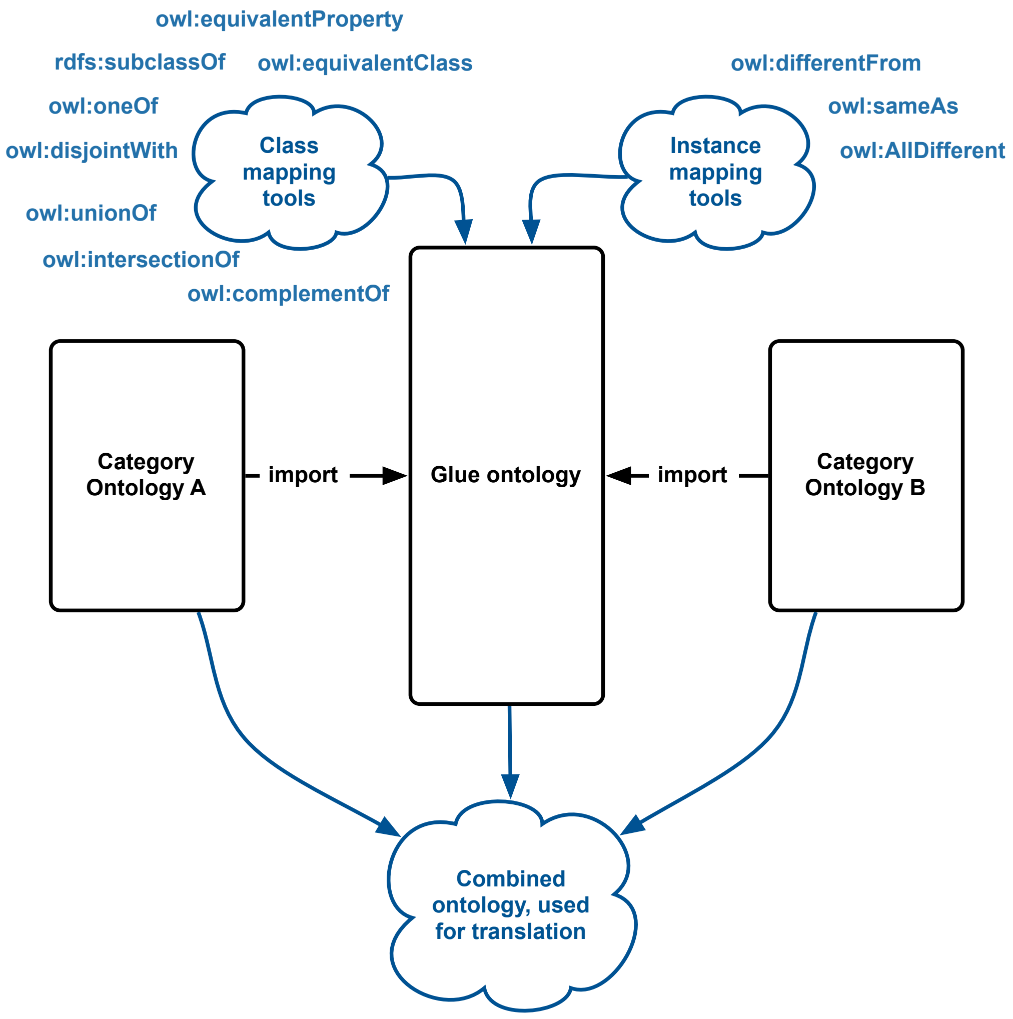


Figure 4.5: Using OWL mapping tools in a glue ontology

tion schemes that describe equivalent or related service categories. Thus, existing service description schemes can be made to coexist, by allowing the discovery of two identical services described using different schemes.

This section defines a framework for explicit translations between ontologies. This framework can be used (as described in Chapter 6) for *implicit* matching of service descriptions.

4.4 Supporting capability-oriented search in OWL-S

While the OWL-S framework promises to enable semantic matching of service descriptions, it does not provide any concrete concepts for publication and discovery of services. It merely claims to be universally applicable due to its declarative descriptions [66]; its support for capability search and its usefulness in service directories are not well explored. Still, as introduced in Section 4.2.1, the features of the OWL-S ServiceProfile are well suited for enabling semantically-enhanced searching of service directories. This section shows how the OWL-S ServiceProfile can be extended to support service discovery better based on its ServiceParameters.

we use the framework defined in Section 4.3 to provide the basis for structuring OWL-S ServiceParameters. Figures 4.6 and 4.7 show how ServiceParameters fit into the schema.

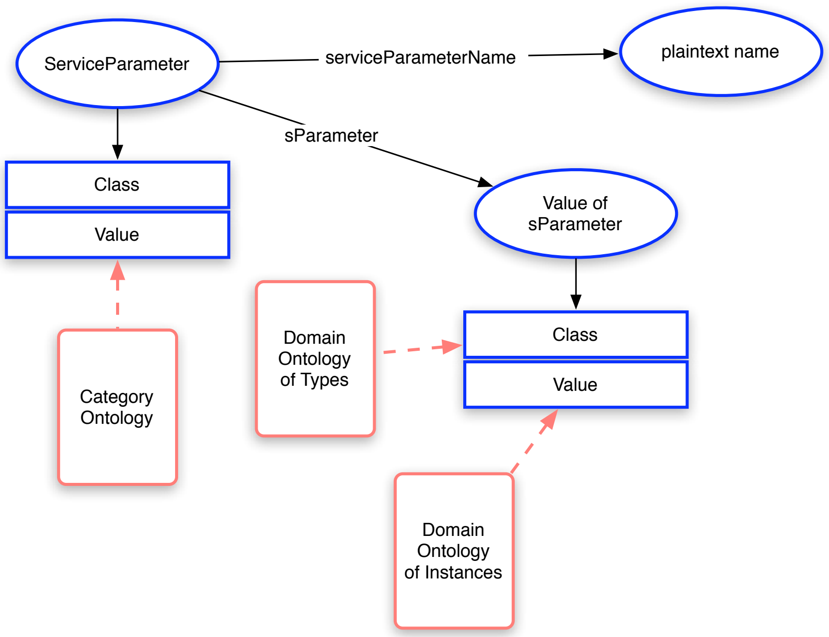


Figure 4.6: Proposed structure of ServiceParameter

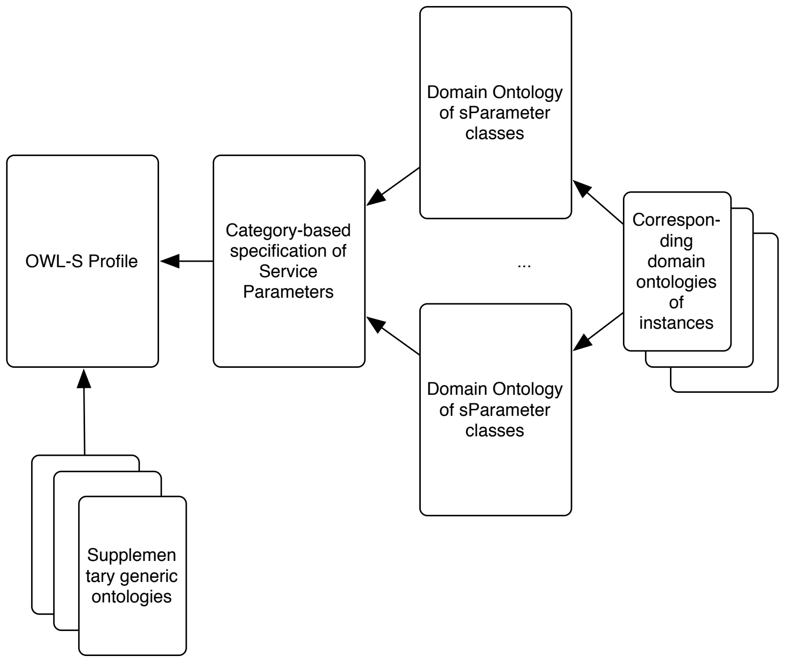

Domain Ontology of sParameter classes
OWL-S Profile
Category-based specification of Service Parameters
...
Correspon-ding domain ontologies of instances
Domain Ontology of sParameter classes
Supplemen tary generic ontologies

Figure 4.7: Ontologies supporting OWL-S Profile in a service directory

4.4.1 Knowledge domain ontologies

The domain ontologies in our schema contain a set of OWL or RDFS classes and their instances. A domain ontology normally treats one subject at a time. However, combined ontologies form a global knowledge base that may be treated as a single unit. Hence the classes and instances can appear in the same ontology files, or spread throughout files, as needed.

While OWL does not disallow ontology inconsistencies, a successful domain ontology should be consistent, i.e. contain no conflicts or ambiguities.

An example of a domain ontology (see Appendix Section A.3.1) would be an ontology of concepts related to network links. An domain ontology of instances (see Appendix Section A.4.1) would then contain a set of pre-defined individual instances that can be used to populate service parameters. For example, an ontology of optical lightpath concepts would contain a "link speed" class, and a pre-defined set of optical link speed instances (OC12, OC48, etc). Other domain-layer ontologies, such as the generic ontology of geographic location (Appendix Sections A.4.2 and A.4.2) can supplement the specific category ontologies for different types of services. Figure 4.8 illustrates how all these knowledge-domain ontologies contribute to the lightpath service description.

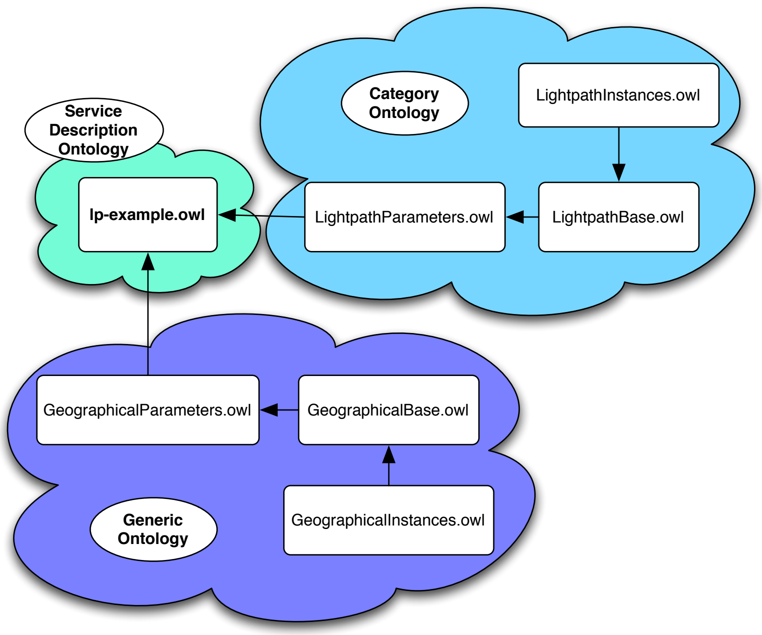


Figure 4.8: Organization of service-related ontologies in the lightpath example

4.4.2 Category ontologies

The category ontology layer introduces OWL-S concepts, and serves as a link between the subject ontology and the OWL-S service description documents. Each service category ontology contains a flat list of parameter classes, all subclassed from the ServiceParameter class, and imports a set of domain ontologies. Each parameter class, in turn, also specifies which classes from the domain ontologies can be used to provide instances for parameter values.

An example of a category ontology would be the collection of ServiceParameter classes relevant to lightpath services (see Section A.2). A lightpath service description might also take advantage of a generic category ontology of service parameter types such as *Location*, *QualityRating* or *ExpiryTime* that could apply to many categories of services.

Figure 4.9 demonstrates subclassing of *serviceParameter* to create a specific parameter type (i.e. a service attribute). This example shows the use of shorthand to refer to ontologies; `&profile;` and `&geo-base;` refer to the full URIs
`http://www.daml.org/services/owl-s/1.1/Profile.owl` and
`http://www.uwaterloo.ca/~jiziembi/owl/GeographicalBase.owl` respectively.

Figure 4.9: An example category-layer definition of a service parameter that subclasses *ServiceParameter*

```
<owl:Class rdf:ID="Location">
    <rdfs:subClassOf rdf:resource="&profile;#ServiceParameter"/>
    <rdfs:subClassOf>
    <owl:Restriction>
        <owl:onProperty rdf:resource="&profile;#sParameter"/>
        <owl:allValuesFrom rdf:resource="&geo-base;#GeographicalRegion"/>
    </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

4.4.3 Service description ontologies (the OWL-S Profile)

We use the OWL-S service description to act as the service description ontology in our hierarchy, but for the purpose of discovery, consider only ServiceParameters of the OWL-S Profile. In Figure 4.10 we see an example of a single lighpath parameter, as it would appear in a service description ontology, alongside its graphical equivalent in the format that we use throughout this thesis. In turn, Figure 4.11 illustrates a full set of the service parameters used in the corresponding OWL-S Profile in Section A.1.

An OWL-S ServiceParameter provides the following information:

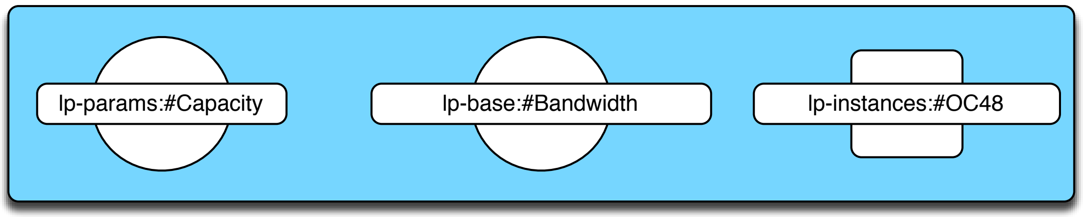
Name	the human-readable name
Type	the OWL class of which the the ServiceParameter is an instance
sParameter Type	the OWL class of which the value of the parameter (sParameter) is an instance
sParameter Value	the value of the parameter

OWL-S has some built-in restrictions on service parameters:

- a `ServiceParameter` has at most one `serviceParameterName`.
- an `sParameter` is restricted to refer to only one concept in some ontology

As part of the new service description schema designed in this thesis, we add the following additional restrictions.

- Each `ServiceParameter` is an instance of a named class in a category ontology.
- The `serviceParameterName` matches its local `rdf:id`.



```
<profile:serviceParameter>
    <lp-params:Capacity rdf:ID="Capacity">
        <profile:serviceParameterName rdf:datatype="&xsd;#string">
            Capacity
        </profile:serviceParameterName>
        <profile:sParameter rdf:resource="&lp-instances;#OC48"/>
    </lp-params:Capacity>
</profile:serviceParameter>
```

Figure 4.10: An example of a parameter of a lightpath service (graphical illustration and XML notation)

- Each `sParameter` is an instance (OWLIndividual) of a named class in a domain ontology.

For parameters with a values from a fixed set (e.g. printer modes), it is especially desirable to define an sParameter class that enumerates these options, to make filling out service templates simpler for the user. Figure 4.12 gives an example of a collection from the OWL Web Ontology Language Reference [72].

In order to guarantee decidability of semantic relationships, and for ease of parsing, all ontologies used in the service directory should conform to the fully-inferable OWL-DL specification [72].

4.4.4 Mapping between ontologies

As outlined in Section 4.3.2 we introduce *glue ontologies* at two layers of the ontology hierarchy.

Knowledge domain layer: Translation must be done either at the level of types or between individuals (class instances). In order for two individuals to be declared equivalent, their classes must be declared to have an intersection.

Category layer: The translation is done at the level of parameter class. For each mapping, a relationship must be defined between the two parameter classes. When

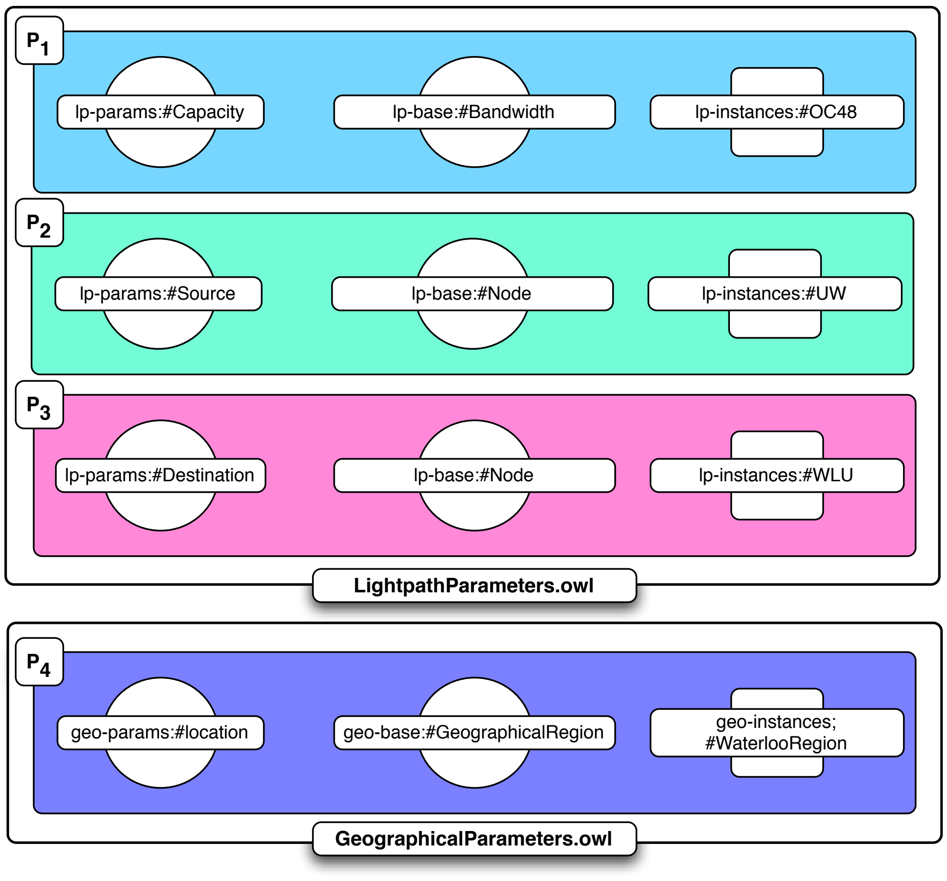


Figure 4.11: An example of a lightpath service description (with shorthand forms representing ontology URIs)

```
<owl:Class rdf:ID="Continent">
  <owl:oneOf rdf:parseType="Collection">
    <owl:Thing rdf:about="#Eurasia"/>
    <owl:Thing rdf:about="#Africa"/>
    <owl:Thing rdf:about="#NorthAmerica"/>
    <owl:Thing rdf:about="#SouthAmerica"/>
    <owl:Thing rdf:about="#Australia"/>
    <owl:Thing rdf:about="#Antarctica"/>
  </owl:oneOf>
</owl:Class>
```

Figure 4.12: An example of an enumerated class from a domain ontology

two parameters classes are declared to be equivalent, and there is a restriction on the types of their values, a relationship must also be declared between the instances at the knowledge-domain layer.

The example glue ontology in Section A.5.2 shows how two different lightpath service description ontologies, `LightpathParameters` and `NetworkLinkParameters` are glued together using subclassing and equivalence, as described above.

The service parameter schema defined in this chapter serves as the basis for populating and searching the service directory. In following chapters, we explore its role in advertisement and querying, as well as the way in which glue ontologies are used to establish semantic relationships between service schemas and service descriptions. First, we take advantage of the modular nature of the service schema to design a simple template-based user interface that uses the ontology hierarchy to build service parameters, and then combines the service parameters into service descriptions and service queries.

Chapter 5

Template-based user interface

This chapter describes the first of the directory components: the interface between the end-user and the directory. The goal of this user interface is to make advertisement and querying as simple as possible from the point of view of the user, while taking full advantage of a powerful back-end search engine.

In keeping with the user-directed, capability-based theme of the service description scheme presented in Chapter 4, the directory aspects presented in this chapter deal primarily with advertising and querying services based on their non-functional service parameters. In this work, an **advertisement** therefore consists of an OWL-S service description document containing a set of service parameters as specified in Section 4.4.3. A **query**, in turn, is made up of a set of similarly-structured service parameters. A previous advertisement can therefore be said to *match* a service query when the query is a subset of the service parameters in the advertised service description.

This approach of querying by subsets of previous advertisements has been used in other service-discovery systems, notably INS/Twine [13], where advertisements are structured as hierarchical attribute-value trees, and queries are created as sub-trees.

The *knowledge base*, which stores ontologies used in service descriptions, is an important supporting element of the service directory. The well-organized layered structure of the knowledge base lends itself well to a template approach to advertisement and querying. Each category-layer ontology can be viewed as a template of service parameters, while the knowledge-domain ontologies are a template for the values of those service parameters. Figure 5.1 illustrates the organization of a service category template as a shallow hierarchy of choices. The obvious uses of a template approach are a simple graphical interface designed for human users, and a clean API that can be used by custom software for automatic generation of service descriptions and queries.

A template-based approach to advertisements and queries helps the end-user create simple queries or add non-functional components to a service description without having to read ontology files or master OWL syntax. The main advantages of a template-based interface are therefore ease of use and transparency to the end-user. However, it

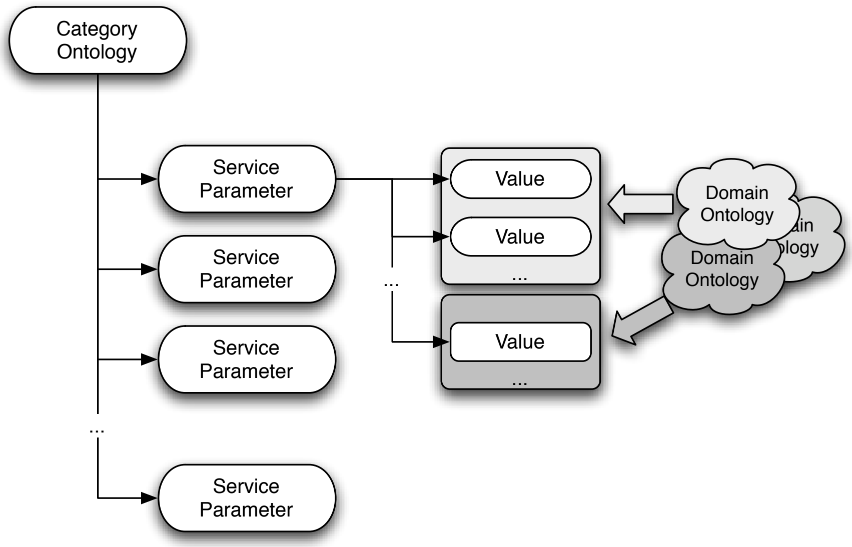


Figure 5.1: Category template presented to the user

is provided merely as a helpful tool that simplifies the user interface and enhances the usability of the knowledge base without affecting its extensibility. The knowledge base can be extended in the following ways: adding a new service category ontology, adding a generic category ontology, adding new parameters to an existing category ontology, or by adding new values to service parameters

One of the advantages of using OWL ontologies is that they can be extended indefinitely by simply importing extra sets of rules into the knowledge base. The above extensions can therefore be performed on-the-fly, during the advertisement/query process. Once the knowledge base has been extended, the user has the option to bypass the interface by creating OWL-S text files independently, or to continue with the template approach. Figure 5.2 shows the flow of generating advertisements and queries using knowledge-base templates.

5.1 Advertisement

To demonstrate the user interaction with the service directory during advertisement (Algorithm 5.1), we present here only those aspects of the user experience that are essential to directory search. As seen in Figure 5.2, most of the user interaction is identical in both advertisement and querying. The main difference is that an advertisement must contain a full service description containing not only service parameters, but also a full OWL-S Profile, Process and Grounding.

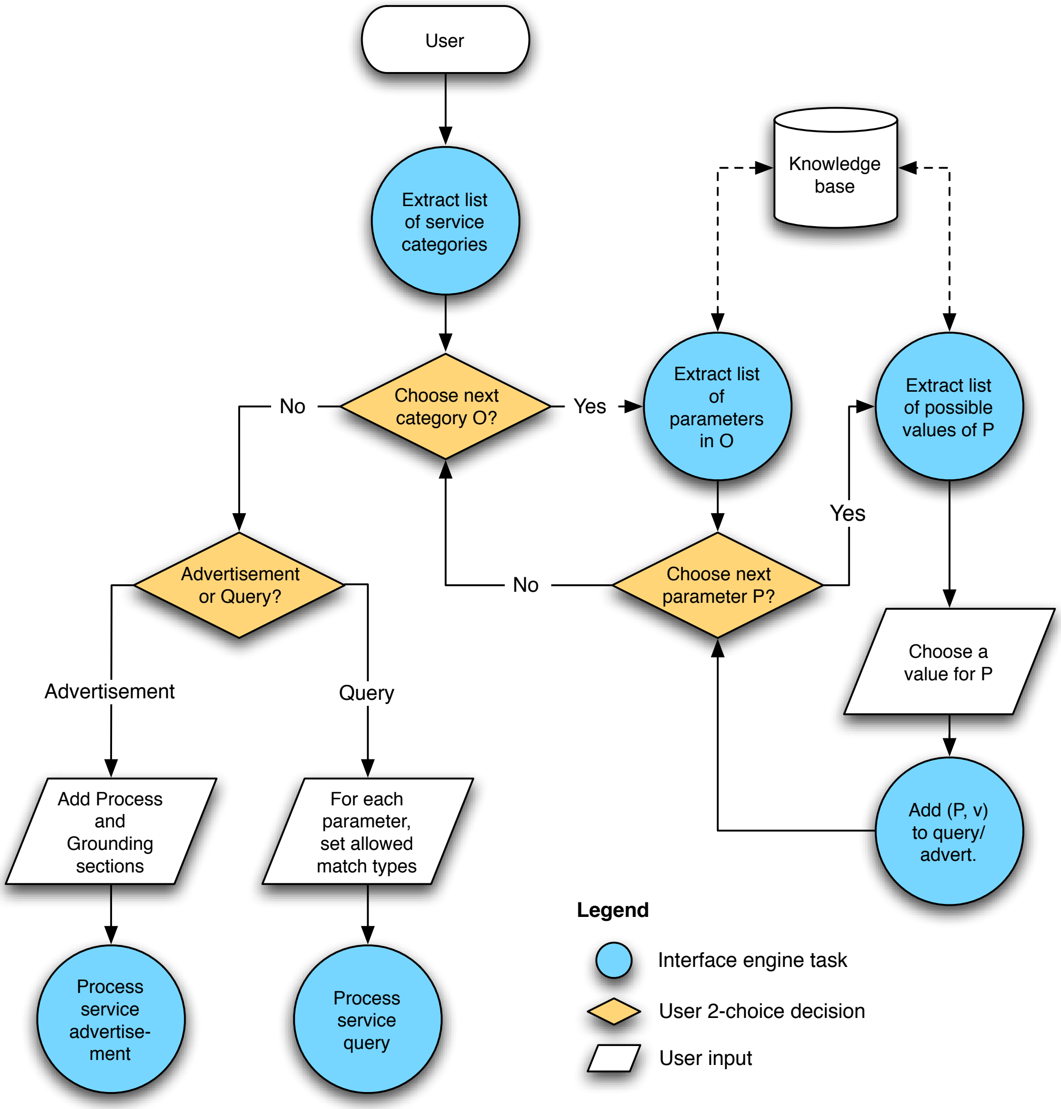

User
Extract list of service categories
Knowledge base
Choose next category O?
No
Yes
Extract list of parameters in O
Extract list of possible values of P
Yes
Advertisement or Query?
No
Choose next parameter P?
Choose a value for P
Advertisement
Query
Add Process and Grounding sections
For each parameter, set allowed match types
Add (P, v) to query/ advert.
Legend
Interface engine task
User 2-choice decision
User input
Process service advertise-ment
Process service query

Figure 5.2: Template-driven user interaction with the service directory

The remaining parts of the OWL-S service description, such as IOPEs, are likely to be identical or nearly identical among similar services belonging to the same category. For example, two different network links may have different bandwidth and access-control policies, but use the same protocol for authentication and sending data. In such a case, the user's work during advertisement is greatly simplified, because it involves re-using existing or pre-packaged Grounding, Process and Profile data, while only needing to fill out a few descriptive parameters in the template.

Other tasks that may occur during advertisement but not during a query may include, for example, setting the advertisement expiry time or access-control policies (e.g. for viewing the advertisement, or for executing the service).

Algorithm 5.1 createServiceAdvertisement

```
Extract from knowledge base: list of service category ontologies
while Adding more parameters to the advertisement do
    User selects          : Option: existing category ontology, or to upload a
            new one?
    if Adding a new category ontology then
        Validate   ontology     Add   new   ontology   to   knowledge   base
                User selects          : Newly-added ontology $O$
    end
    else
        User selects          : Existing ontology $O$
    end
    Extract from knowledge base: Category template of $O$
    foreach Type $C$ of service parameter do
        Display to user       : List of possible parameter values $v$
        User selects          : Parameter value, or null
        if selection not null then
            Generate          : Service parameter $P$ of type $C$ with value
                $v$
            Add $P$ to advertisement
        end
    end
end
User fills in the rest of the advertisement in OWL-S format
```

5.2 Querying

The basic query process, presented in Algorithm 5.2 follows the same steps for generating service parameters. While service parameters are being created, a user also specifies the *match types* which determine the extent of semantic expansion during queries. A fully-featured service directory would additionally allow for piggybacking additional query types beyond those supported by the indexing scheme described in detail in Chapter 7. These query types, (for example range queries, or those containing negation or alternatives) would then be processed locally by the database storing service descriptions.

Algorithm 5.2 createServiceQuery

```
Extract from knowledge base: list of service category ontologies
while Adding more parameters to the advertisement do
    User selects : Option: existing category ontology, or to upload a new one?
    if Adding a new category ontology then
        Validate ontology Add new ontology to knowledge base
        User selects : Newly-added ontology $O$
    end
    else
        User selects : Existing ontology $O$
    end
    Extract from knowledge base: Category template of $O$
    foreach Type $C$ of service parameter do
        Display to user : List of possible parameter values $v$
        User selects : Parameter value, or null
        if selection not null then
            Generate : Service parameter $P$ of type $C$ with value $v$
            Add $P$ to query Select allowable match types for $P$ from $\{EXACT, SUBSUME, RELAX\}$
        end
    end
end
```

In our lightpath ontology example, we build a query (see Figure 5.3) which is a subset of the service description in Figure 4.11 and asks 11Find a lightpath service with an OC48 capacity and located within Waterloo Region". The query uses two category-layer ontologies: `LightpathParameters.owl` and `GeographicalParameters.owl` (represented by `lp-params` and `geo-params`) to build the two-parameter query. The value types of the queries are taken from the knowledge-domain ontologies of types: `LightpathBase.owl (lp-base)` and `GeographicalBase.owl (geo-base)`, and the values are taken from knowledge-domain ontologies of instances `LightpathInstances.owl (lp-instances)` and `GeographicalInstances.owl (geo-instances)`. This query example also demonstrates the choice of allowable match types: we allow only $EXACT$ matches for the first

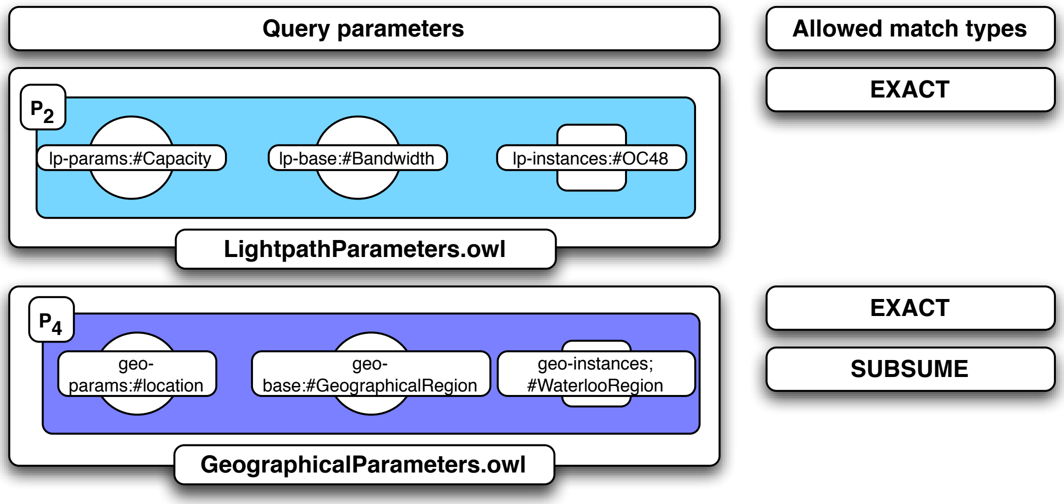


Figure 5.3: Example of a query that uses two category-layer ontologies

parameter, `Capacity`, and both *EXACT* and *SUBSUME* match types for the second parameter, `Location`.

5.3 Benefits of a template-based approach

The main benefit of the user interface presented in this chapter is its simplicity. A template-based approach is intuitive to the end-user. For the service provider, it is easier than having to write the service-description document by hand. Similarly, for a service-seeking user, it is easier than learning a sophisticated query language such as XQuery [12], XPath [73] or SQL [11]. Because the knowledge base of templates (ontologies) lends itself well to extensibility by dropping-in new templates, this approach remains simple without being restrictive.

Furthermore, this template-based user interface is transparent: the ordering of steps and the set of choices presented to the user reflects the underlying structure of the knowledge base. This is educational, and reduces the learning curve required by an end-user to write extensions to the knowledge base.

Lastly, the hierarchical processing of templates translates well to a simple web-based interface, whether graphical or text-based. This kind of interface can be used in a web-browser on a desktop computer, mobile computing device or even a cell phone, since it does not require much text input, but merely a sequence of choices.

Once service advertisements and queries have been constructed as above, they are ready to be used in a service directory. The directory can then search for services using either the simple “query as a subset of matching advertisement” relationship, or the more complex semantic matching algorithm presented in the following chapter.

Chapter 6

Semantic matching of service descriptions

Exploiting semantics in service discovery is in in some ways easier and in others, more difficult than exploiting them in a knowledge-base semantic web application (for example, Wordnet [57]). On one hand, the structured format and restricted vocabulary of service descriptions lend themselves well to automation of parsing and translation. On the other hand, while semantic-web content is targeted mainly towards open-ended, human-directed search, service matching is ultimately targeted for machine use, and needs to be more precise and rigorous. In this chapter, we explore how semantic matching can enhance service discovery.

In Chapter 4, we use service description to take advantage of the narrowing aspect of web semantics (see Section 4.2.1). This chapter in turn adds the benefit of the broadening aspect, by designing the semantic search engine of the service directory. This search engine is a set of algorithms that take advantage of glue ontologies to create *semantic synonyms* used for query expansion. In translating the user's queries, we hope to find services whose service descriptions may not be a literal match for the query, and would not be matched by standard keyword-based engines, but which *semantically* match the query. In short, the goal of the search engine is to broaden the search to include as many equivalent services as possible, while avoiding false hits.

The semantic matching algorithms presented below build on the framework described in Chapter 4. This means that we assume the existence of a service directory containing a set of service advertisements and a collection of knowledge domain and category ontologies. There are two choices for when to perform semantic matching of service descriptions:

At time of advertisement: Each service description is translated into a set of equivalent descriptions and then advertised. One could either translate each advertisement to a standardized format (this would reduce advertising cost) or – which is much more expensive, but also more realistic – translate it into several equivalent advertisements. In the latter case, it likely makes more sense to translate an entire service description scheme as a whole rather than to separately create synonyms of each parameter. This approach is less flexible, but it avoids the unnecessary overhead incurred by translating each part of the description separately.

Advertisement-time semantic matching on a distributed directory can be used to create a semantic overlay. Documents inserted into the directory can be linked based on a similarity metric, so that similar advertisements are located logically close together (few hops) in the overlay. The semantic overlay can be constructed by advertisement-time semantic expansion (by making several semantically-related advertisements), or built into the advertisement routing algorithm (by grouping semantically-related advertisements together).

At time of query: Each query results in the creation of a partial description – it creates search parameters that may have been previously advertised in matching service descriptions. As in advertisement-time translation, the creation of new search parameters using ontological rules can be performed in two ways.

1. For each search parameter, generate a set of matching parameters.
2. For an entire set of search parameters, generate an equivalent set by template-level translation (in each translated description, all parameters belong to the same ontology).

Matching each parameter is more flexible, and is likely to return a broader set of results, but may generate a prohibitive number of combinations. If service descriptions tend to use well-defined templates, the second option makes more sense, since service descriptions are then unlikely to mix parameters from different ontologies. However, template-based translation is less flexible and it is possible that the original search may not return any results at all.

In this work, we choose to use query-time matching. Query-time matching removes the need to guess ahead of time which service-description schemas will be used for querying in the future. Instead, we can translate queries based on the already advertised schemas. Query-time matching also avoids the overhead of storing translated advertisements which may never be retrieved by users.

6.1 Types of OWL semantic matching

The Web Ontology Language not only works well as a service description language, it also serves as a foundation for semantic reasoning. Using an OWL *reasoner* (software used for inference) we can discover relationships among OWL classes and their instances. This section briefly describes the kinds of class- and instance-based semantic mapping available in OWL, focusing on those options most useful to service discovery.

6.1.1 Mapping of classes

Given two OWL classes, $C1$ and $C2$ we can describe four kinds of binary relationships:

EXACT: $C1 \equiv C2$ ($i \in C1 \Leftrightarrow i \in C2$)

SUBSUME: $C1$ is a subclass of $C2$ ($i \in C1 \Rightarrow i \in C2$)

RELAXED: Inverse of SUBSUME: $C1$ relaxes $C2 \Leftrightarrow C2$ subsumes $C1$

DISJOINT: $C1$ is disjoint from $C2$ ($C1 \cap C2 = \emptyset$)

This mapping can be explicit (using *rdfs:subclass*, *owl:equivalentClass*, *owl:unionOf* and *owl:intersectionOf*), or inferred from other properties. Note that all of the above relations are **transitive**; EXACT (class equivalence) and DISJOINT are also symmetric. The transitivity adds a great deal of power to semantic inference, since it allows subsumption and relaxation relationships to be derived from complex chains of rules. Since our semantic matching engine deals with only two service description at a time, we ignore n-ary class relationships such as UNION_OF or INTERSECTION_OF.

OWL can have named classes, and anonymous classes that are defined by their restrictions. An OWL class can be identified in terms of its relationship with an anonymous class. For example, OWL allows the class *mammals* to be defined as a subclass of all animals with the property "has fur". Such aonymous classes are used in service descriptions primarily to define restrictions on category-layer classes.

6.1.2 Mapping of instances

The OWL primitives for mapping class instances (OWL individuals) to each other consist primarily of strict-equivalence relationships: *owl:sameAs*, *owl:differentFrom* and *owl:allDifferent*. This mapping can be done explicitly, by specifying these relationships in an ontology file, or implicitly, by deriving these properties from other ontological relationships. For example, *owl:sameAs* can be derived by transitivity, or by inferring that both instances have the same InverseFunctional property. [1]

Besides simple equivalence, more complex relationships between instances can be expressed by defining custom OWL properties, describing the type of these properties (e.g. *owl:transitiveProperty*, *owl:functionalProperty*) and ascribing them to individuals. Using custom properties ("verbs") to create and infer relationships between individual entities is the biggest expressive power of OWL. These relationships are concept-specific and cannot be generalized; for example, *locatedIn* is a transitive property useful for describing geographical locations, but cannot be applied to payment options of an Internet service.

[1]If an OWL property P is tagged as InverseFunctional then for all x, y and z, $P(x, z)$ and $P(y, z)$ implies $x = y$ [27]

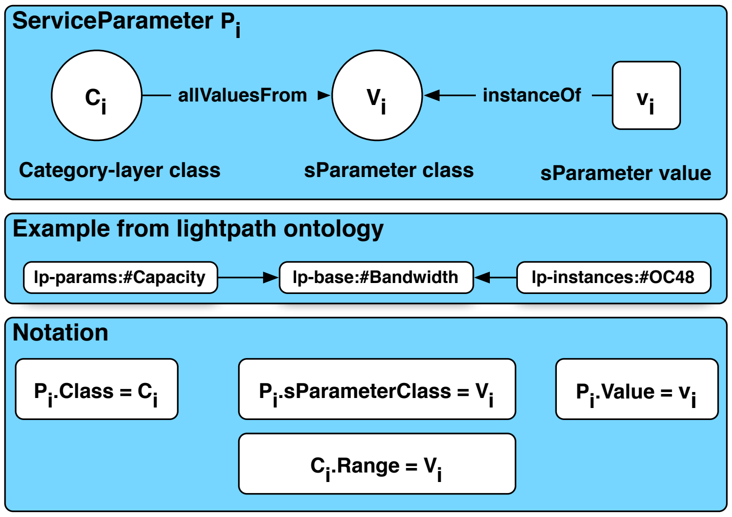


Figure 6.1: The elements of a service parameter P_i used for semantic matching

6.2 Matching service descriptions

Before discussing how to expand searching for services using synonyms and semantic inference, it makes sense to first define what it means for two services to have matching descriptions. In this work, we assume OWL-S service descriptions structured as described in Section 4.4 and restrict the matching of service descriptions to non-functional OWL-S ServiceParameters. Two service descriptions can be said to match if they have matching ServiceParameters. We begin by defining the relationships between service parameters in terms of their components.

Suppose we have two service parameters as : $P_1 \in O_1$ and $P_2 \in O_2$, where O_1 and O_2 are P_1's and P_2's "parent" category-layer ontologies. The match between P_1 and P_2 is determined using three pieces of information contained within the parameter definition: the category-layer class of the parameter, the knowledge-domain class of the value of the parameter (determined by the *allValuesFrom* or *someValuesFrom* restrictions of the category-layer class), and the value of the parameter (see Figure 6.1).

We follow with a description of each of the three types of semantic matches using class and instance relationships of the components of service parameters.

6.2.1 EXACT match

An exact match is the only one that can be done generically, since the *sameAs* and *equivalentClass* relationships are the only well-defined built-in OWL properties of individuals.

Two ServiceParameters, P_1 and P_2 are said to be an exact match if the following conditions are met:

1. $P_1.Class \equiv P_2.Class$
2. $P_1.Value \equiv P_2.Value$

6.2.2 SUBSUME match

Matching by subsumption is trickier than matching by equivalence. It is necessary to define a knowledge-domain layer property *Prop* such that $Prop(I_1, I_2)$ in some way expresses a subsumption ("containment") relationship between class instances. An example of such a property in an ontology of geographic location is "contains" – for example "Waterloo Region `contains` Cambridge".

Two ServiceParameters, P_1 and P_2 are said to be a match by subsumption if the following conditions are met:

1. $P_1.Class \equiv P_2.Class$ or $P_1.Class \subseteq P_2.Class$
2. $P_1.sParameterValue$ subsumes $P_2.sParameterValue$

   (that is $Prop(P_1.sParameterValue, P_2.sParameterValue)$

   where *Prop* is a relaxing property).

6.2.3 RELAXED match

A relaxed match is the reverse of a subsumption match. As above, expressing this relationship requires defining a relaxing property between the instances of the sParameter class. An example of a relaxed property is *locatedIn* in context of a geographic location – for example "Cambridge `locatedIn` Waterloo Region".

Two ServiceParameters, P_1 and P_2 are said to be a relaxed match if the following conditions are met:

1. $P_2.Class \equiv P_1.Class$ or $P_2.Class \subseteq P_1.Class$
2. $P_2.sParameterValue$ subsumes $P_1.sParameterValue$

   (that is $Prop(P_1.sParameterValue, P_2.sParameterValue)$

   where *Prop* is a relaxing property).

One of the great features of OWL is that subsuming or relaxing properties do not have to be originally defined in knowledge-domain ontologies. Glue ontologies can be used to either create new properties, or mark existing properties as being of type "subsuming" or "relaxing". We see an example of this in the sample glue ontology in Appendix Section A.5.2 which uses the subsuming/relaxing property types defined in Appendix Section A.5.1 to mark existing properties *locatedIn* and *contains* as subsuming and relaxing, respectively.

6.3 Semantic matching algorithms

In this set of algorithms, a query is simply a partial service description (a set of OWL-S ServiceParameters) created from a category template, as described in Chapter 5. Given a user-specified query, we use the principles behind matching service descriptions from Section 6.1 to **build a set of equivalent service descriptions** that can be used in directory search. The algorithms presented below depend on building *parameter synonyms* and *query synonyms*. The exact definition of *synonym* depends on whether EXACT, SUBSUME or RELAXED matching is used (see Section 6.2). We assume that for each parameter, the types of matches allowed in building synonyms (i.e. a non-empty subset of {EXACT, SUBSUME, RELAXED}) have been chosen previously by an end-user. Hence, given a service parameter P, we denote as its *synonym* another parameter P' such that P' *matches* P. Similarly, a query q' is said to be a *synonym* of a query q if for each service parameter $P \in q$, there exists a parameter $P' \in q'$ such that P' *matches* P. In a general sense, a synonym of a query q is a query that will return services with equivalent properties and functionalities to those returned by q.

This section presents a set of algorithms for building a knowledge base within a directory, for generating parameter synonyms and for combining parameter synonyms to build full query synonyms.

6.3.1 Setting up the directory knowledge base

The semantic search engine uses two main tools:

Knowledge base: a database of ontologies used for service description. As outlined in Chapter 4, this database consists of knowledge-domain ontologies, category ontologies and glue ontologies.

Semantic matching algorithms: the algorithms which use the knowledge base to create semantic synonyms of queries.

Each directory node contains a database of knowledge-domain, category and glue ontologies. The knowledge base provides the organizing principles for the collection of service descriptions in the directory. While the indexing engine and the storage engine concern themselves with locating the data in the directory, the knowledge base serves as the "brains" that determine how well the semantic engine can analyze the data. The richer the relationships within the knowledge base, the "smarter" the semantic engine is in creating semantic synonyms. In that respect, a knowledge base should remain a flexible entity such that its set of ontologies can be dynamically expanded. The storage and maintenance of the knowledge base are tackled in Section 7.3.3. In the meanwhile, Algorithm 6.1 serves as a guideline for consolidating the various levels of ontologies into a single unit.

Algorithm 6.1 setupDirectoryNode(P)

```
Read in knowledge-domain ontologies of types
Read in knowledge-domain ontologies of instances
Read in category-layer ontologies of service parameters
Extract the subsuming and relaxing properties
```

The biggest drawback to using formal ontologies rather than a statistical, plaintext-based approach, is that all ontologies must be manually created. If possible, it is best to take advantage of existing ontologies. If there are no suitable existing ontologies for the service type we are trying to describe, it is necessary to build the ontologies from scratch. The first step is choosing the knowledge-domain ontologies. If possible, it is important to choose commonly-used ontologies, to reduce the effort needed in translating them later; many knowledge-base ontologies are already being developed by research bodies [74]. The next step is choosing relevant parameters to include in the category-layer ontology and linking them to the knowledge-domain ontologies. For universal parameters such as cost or location, it is a good idea to use parameters already existing in generic category-layer ontologies. Finally, as new ontologies are added to the knowledge base, they must be incorporated using glue ontologies.

While the creation of a knowledge base is certainly a complex task, once ontologies are created, they can be shared among service providers, like in the case of service-description schemas in current service-discovery systems such as UPnP [2]. The effort spent on creating and "gluing" category-layer technologies is validated by the subsequent ease in creating service descriptions. Since there are likely to be many more advertised services than the corresponding service categories, and since the number of category synonyms that need to be glued together should also be relatively small, building a well-structured knowledge base is a worthwhile effort.

6.3.2 Algorithm: Building parameter synonyms

In order to create query synonyms, we first break up the query into its component service parameters. Algorithm 6.2 takes as input a single service parameter P (structured as shown in Figure 6.1), and uses the knowledge base to generate a set of synonyms for P, as illustrated by Figure 6.2 to find the synonyms for each parameter.

6.3.3 Algorithm: Building query synonyms

Once we have a set of parameter synonyms for the original query, we combine them to build query synonyms. The technique for combining the parameters must be chosen carefully. Given the original query $q = \{P_1, \ldots, P_n\}$, generating every possible combination of service parameters would result in $|Synonyms(P_1)| \cdot |Synonyms(P_2)| \cdot \ldots \cdot |Synonyms(P_n)|$ query synonyms, which is exponential in the number of parameters.

However, knowing that queries are normally created using a single category ontology

Algorithm 6.2 findParameterSynonyms(P)

```
Input  : $P = \{C, V, v\}$
Input  : $AllowedMatches(P) \subseteq \{EXACT, SUBSUME, RELAX\}$
Output: Synonyms(P)
Let $C$ := $P.Class$   Let $V$ := $P.sParameterClass$   Let $v$ := $P.Value$
if $EXACT \in AllowedMatches(P)$ then
    foreach $C'$ such that $C'$ is an equivalentClass of $C$ do
        Let $V'$ := $C'.Range$   foreach $v' \in V'$ such that $v'$ is sameAs $v$ do
            Create a new service parameter $P' = \{C', V', v'\}$   Add $P'$ to Synonyms(P)
        end
    end
end
if $SUBSUME \in AllowedMatches(P)$ then
    foreach $C'$ such that ($C'$ is an equivalentClass of $C$ or $C'$ is a subclass of $C$ do
        Let $V'$ := $C'.Range$   foreach $Prop \in SubsumingRelations(V)$ do
            foreach $v' \in V'$ such that Prop(v, $v'$) do
                Create a new service parameter $P' = \{C', V', v'\}$
            end
        end
    end
end
if $RELAXED \in AllowedMatches(P)$ then
    foreach $C'$ such that ($C'$ is an equivalentClass of $C$ or $C$ is a subclass of $C'$ do
        Let $V'$ := $C'.Range$   foreach $Prop \in RelaxingRelations(V)$ do
            foreach $v' \in V'$ such that Prop(v, $v'$) do
                Create a new service parameter $P' = \{C', V', v'\}$
            end
        end
    end
end
return Synonyms(P)
```

(see Section 5.1), we can significantly reduce the number of query synonyms, and increase the chances that each query synonym will result in meaningful hits. Therefore, we group parameter synonyms according to their parent category, and generate all possible combinations of parameters within each category ontology. This process is illustrated in Figure 6.3. Suppose that P_1, P_2 and P_3 are three query parameters, all originating from category ontology A, and that category ontologies B, C and D have been found to contain synonyms for these parameters. For example, B contains three synonyms for P_1, two synonyms for P_2 and one for P_3. We combine these synonyms within B, generating $3 \cdot 2 \cdot 1 = 6$ query synonyms from B. On the other hand, because C does not contain a synonym for P_2, we do not use it in generating query synonyms. The crucial aspect of this algorithm is that if two query parameters were taken from the same category ontology, then their synonyms in any resulting query synonym will also be from a common category ontology (e.g. we would not combine a synonym of P_1 from B with a synonym of P_2 from D).

Each combination of service parameters considered a *query synonym*. Since we want to retrieve only those results which match all of the query parameters, we discard those categories that do not have synonyms for one or more of the original parameters. Algorithm 6.3 formalizes the process for generating query synonyms for a query that is built using a single category ontology. For those queries which use additional generic category-layer ontologies, this process is repeated for each of those generic ontologies, and combinations are generated according to the same principles as outlined above.

Algorithm 6.3 generateQuerySynonyms($q = \{P_1, \ldots, P_n\}$)

```
Let $Q$ be the resulting set of synonym queries  foreach service parameter $P_i \in q$ do
    let Synonyms($P_i$) := findParameterSynonyms($P_i$)
end
/*Let $\Omega$ be the set of category-layer ontologies $O$ that have generated
  synonyms for the service ontology                                          */

$\Omega = \{O \mid \exists\, C' \in O$ such that $P' \in Synonyms(P)$ for some $P' \in C'$ and $P \in \{P_1, \ldots, P_n\}\}$

    /*To simplify, if $P' \in C'$ and $C' \in O$ then let us write $P' \in O$      */
foreach $O \in \Omega$ such that $O \cap Synonyms(P) \neq \emptyset \forall P \in \{P_1, \ldots, P_n\}$ do
    Create all possible queries $q = \{P'_1, \ldots, P'_n\}$ such that $P'_i \in O$ and $P'_i \in Synonyms(P_i)$
    $\forall i \in \{1 \ldots n\}$  Let $Q_O$ be the set of all queries generated for ontology $O$.  Add $Q_O$ to
    $Q$
end
return $Q$
```

Having built our example query in Figure 5.3, we now use the equivalence and subsumption relationships in our knowledge base to generate query synonyms. In the case

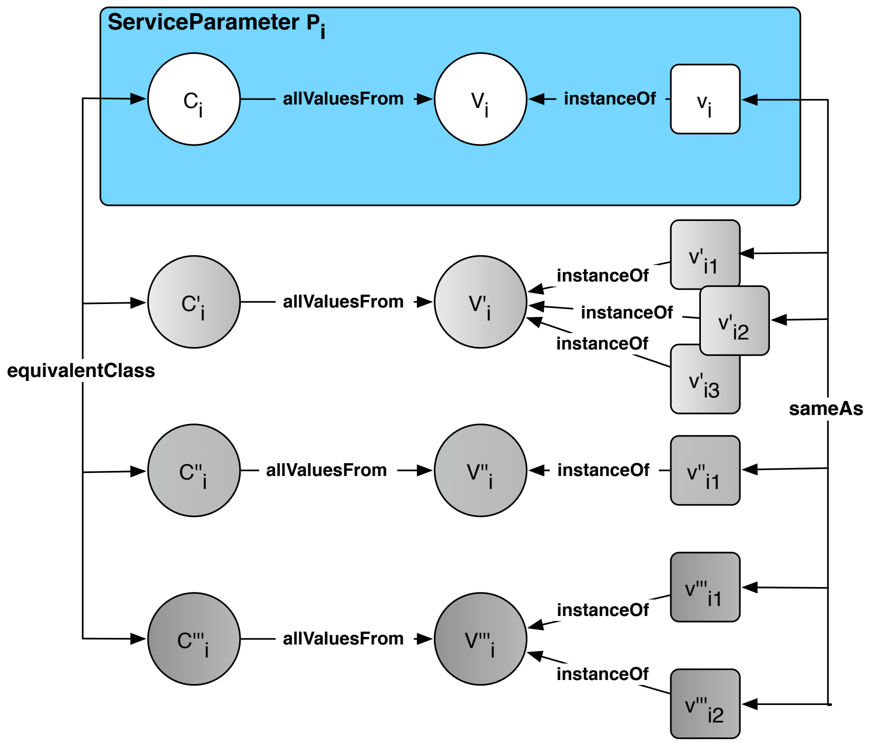


Figure 6.2: Finding synonyms for a single parameter

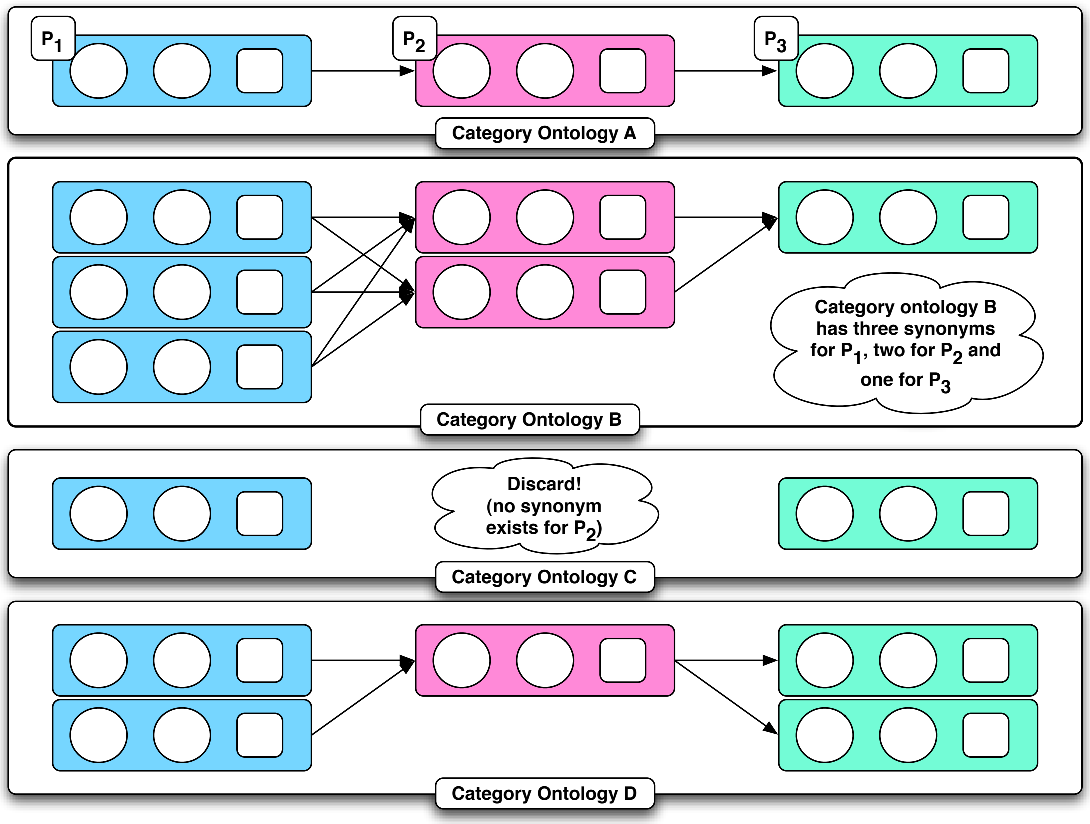

P1
P2
P3
Category Ontology A
Category ontology B has three synonyms for P1, two for P2 and one for P3
Category Ontology B
Discard! (no synonym exists for P2)
Category Ontology C
Category Ontology D

Figure 6.3: Combinations of synonyms for query parameters P_1, P_2 and P_3 constructed by Algorithm 6.3

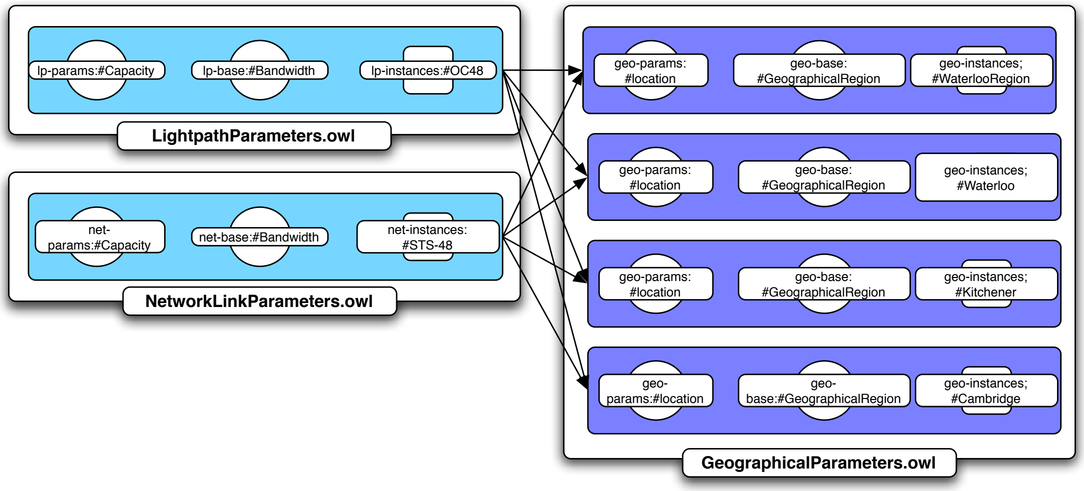


Figure 6.4: Example of generating query synonyms

of the first parameter, `Capacity`, the semantic matching algorithm generates a parameter synonym from a different domain ontology, `NetworkLinkInstances.owl`, using the glue ontology. For the second parameter, `Location`, the subsuming property, *locatedIn* generates three synonyms from the original knowledge-domain ontology. Combining these parameter synonyms withthe original query parameters generates $2 \cdot 4 = 8$ query synonyms, as illustrated in Figure 6.4.

6.3.4 Algorithm: Searching the directory

Once query synonyms are generated, a separate search is performed for each of these queries, and the results are consolidated. Once full service descriptions are retrieved, a separate refining must be performed to select only those services which can be invoked by the client. This post-processing can be using a process-level matchmaker such as the OWL-S Matchmaker [29], which is being developed to match semantic Web services using functional parameters.

Algorithm 6.4 semanticSearch(q)

```
/*Find all the query synonyms                                  */
Let $Q = generateQuerySynonyms(q)$ foreach $q' \in q \cap Q$ do
    perform a search on $q'$ in the service directory
end
Refine search further using functional parameters
```

6.3.5 Complexity analysis of the semantic matching algorithm

Before deploying the above searching algorithm in a real-world scenario, we must first determine whether it is not prohibitively expensive in terms of search operations. The search algorithm should also avoid performing extraneous work that is not likely to elicit meaningful results.

The number of search operations performed on the directory in Algorithm 6.4 depends on the number of query synonyms generated in Algorithm 6.3. Let $q = \{P_1, \ldots P_n\}$ be the original query and let $Synonyms(q) = \{q_1, \ldots, q_k\}$ be the set of query synonyms. The complexity of the "glue" between ontologies is the true factor that determines the number of query synonyms; therefore let us examine the queries from the point of view of Ω, as introduced in Algorithm 6.3. Let Ω be the set of category-layer ontologies that have generated synonyms of q, so that Ω consists of ontologies O such that $O \cap Synonyms(q) \neq \emptyset$, that is, $q_i \subset O$ for some $q_i \in Synonyms(q)$.

$$|Synonyms(q)| = \sum_{O \in \Omega} \prod_{i=1}^{n} |Synonyms(P_i) \cap O| \tag{6.1}$$

Equation 6.1 shows that there are two main factors in determining $|Q|$:

1. The number of category-layer ontologies which generate full query synonyms

$$\{|O \in \Omega | O \cap Synonyms(P_i) \neq \emptyset \forall i \in 1, \ldots, n|\}$$

2. the number of synonyms for each P_i in each ontology ($|Synonyms(P_i) \cap O|$ for each $O \in \omega$ and $1 \leq i \leq n$).

The ontologies that generate full query synonyms would be created independently by different service providers, and must all be reachable from the original query ontology by a "chain" of glue ontologies. It is not unreasonable to speculate that when creating a new set of service description ontologies, a service provider would make a reasonable effort to locate existing ontologies. Moreover, glue ontologies are likely to be created only between well established service ontologies. We can therefore expect that the number of ontologies would be relatively small: on the order of a few to a few tens rather than hundreds or thousands.

The second factor, the product of the number of synonyms for each parameter, is more important in this equation, because it is a product, and because it depends on the complexity of the underlying knowledge-domain ontology. If the ontology contains a deep structure of terms connected by equivalence, subsumption or relaxation properties, this number could be very large. However, because the synonyms created by the semantic matching algorithm are based on explicitly stated relationships, the final amount of semantic expansion can be expected to be lower – and the synonyms themselves of better quality – than in systems where search is expanded by implicitly-derived semantic similarity relationships [51] or [54]. The size of the second factor can be kept small by revising Algorithm 6.3 to explicitly restrict the number of synonyms created from each service parameter, and by using equivalence search in place of subsuming/relaxing search.

The semantic matching algorithms presented in this chapter take advantage of the design of the service description and of the organization of the knowledge base to expand queries during the service discovery process. The query expansion is performed predictably, inferring from explicit semantic relationships defined in the knowledge base ontologies, rather than from approximative metrics derived from keyword-based dictionaries. Query synonyms derived from this expansion have therefore a high probability of being true synonyms of the query, which reduces the probability of “false hits” during the service discovery process.

Having covered query creation and expansion, we now progress to handling both service descriptions and queries in the context of the distributed directory. The following chapter introduces the design of the service directory and discusses its mechanisms for processing service advertisements and queries.

Chapter 7

Distributing the service directory

In this chapter, we address the *indexing engine* and the *data storage engine* of a service directory. We draw a distinction between the two components by elaborating on their brief definitions presented in Chapter 1, and discuss how to enhance both by distributing them in a large-area network. Section 7.1 presents the goals for the design and implementation of this large-scale service directory. Section 7.2 outlines the design of an indexing scheme to enhance the performance of a directory search, and Section 7.3 illustrates methods for distributing and maintaining the index, the service description data and the knowledge base. We conclude this chapter by addressing design issues related to indexing and data distribution in Section 7.4 and by discussing the complexity of advertisements and queries in Section 7.5.

7.1 Goals for a large-scale service directory

The requirements of a large-scale service directory are different from those of a local database or a local service-discovery system. A local discovery system may not even require a dedicated directory structure; service advertisements and queries can be simply broadcasted or multicasted among participating nodes. In a wide-area network, however, this type of approach would generate an unmanageable amount of network traffic and delay. A service-discovery system suited for a wide-area network would therefore need to scale accordingly without incurring long delays or crippling the network. Moreover, the directory design must take into consideration the multi-domain aspect of our target infrastructure: administrative domains prefer to be self-governed, and may be unwilling to surrender access control to their services or service-description data, while still making it possible to discover services across domain boundaries. Therefore, we derive the following design goals for the distributed multi-domain service directory.

1. **Reasonably fast service registration, very fast querying**. We assume that there will be fewer services being registered than users of these services, therefore

search time should be optimized. Querying should aim to take about as much time as a web search using a modern search engine, i.e. at most a few seconds. The speed of service registration need only be this fast if it involves immediate feedback to the entity (or person) who registers the service.

2. **Scalability.** An Internet-scale directory should be able to register a very small or an arbitrarily large number of services without significant impact on its responsiveness (speed of advertisement and querying).

3. **Completeness of query results.** A query should be capable of returning all, or at least a minimum set of matching services if they exist in the directory, regardless of where the services are located in relation to the querying entity.

4. **Independence of domains.** A service directory should allow the participation of service providers and end users from multiple administrative domains, without the need for a central authority for processing service registrations and queries.

5. **Service providers retain responsibility for their data.** A service provider may advertise portions of its service descriptions to the world at large, but retain the ability to restrict the access to full descriptions (for example, the application interface). It is hence preferable that service descriptions be stored under the control of the service provider.

In order to achieve the goals of responsiveness and scalabilty, the service directory cannot rely on a single centralized database. A service directory can simply replicate its data across different physical locations, or distribute the data over the network and tie it together with some type of index. Note, however, that a physically distributed directory can appear to be logically centralized; i.e., the directory has a single access point, and the directory stores the same data from the point of view of any user in the network. This is common for both replicated and schema-based distributed directories, where all directory nodes know all the schemas used to describe directory data. In this chapter, we design a directory where both the index and the data are distributed over the network, and the service directory may appear to have different contents depending on the set of schemas known at the endpoint from which it is accessed. This type of flexible approach, which does not assume a unified schema across the network, allows different administrative domains to use their own schemas or update existing ones, and to propagate them to other nodes at their leisure.

The administrative structure of the Internet is in constant flux: companies and organizations come and go. For a service directory to function in an Internet-scale environment, it needs to be flexible, extensible and resilient to changes; it is unrealistic to supose that it would be managed by a single administrative entity. Therefore, it is desirable that this type of service directory be organized in a loosely-coupled peer-to-peer architecture.

General indexing	Each triple functions as an atomic unit
Hash-based	Each of the components has a distinct role; they cannot be
indexing	re-arranged in a different order to create a different triple
String hashing	Triples use URI; only exact match is required
Bloom filters	Queries are performed as tests of membership of all triples in a service description

Table 7.1: Suitability of service description scheme for hash-based indexing and Bloom-filter-based indexing

7.2 Indexing engine

The main purpose of an index is to reduce the number of direct accesses to the data while searching. Given a large-scale service directory, it is infeasible to do a full-text search of its database. While a good index is important in a centralized directory where it aims to reduce the number of disk accessess, it is even more important in a distributed directory, where a data access may mean one or more network accesses, which are normally far more expensive than local disk accesses or storage area network (SAN) accesses.

An index uses a small amount of information extracted from the data to facilitate access to this data. In order for an index to be successful, this information needs to correspond, in some way, to the queries used to retrieve the data. The main question in choosing an indexing scheme is, therefore, which information to use in an index, and how to organize it within the index. Since service descriptions are text-based, we focus on index techniques that are suitable for structured text data and can be adapted for use in a distributed directory.

The properties of a suitable indexing scheme depend on the properties of the data and properties of the queries. In the case of the service description scheme presented in Chapter 4, both the data and the queries are presented using sets of service parameter triples of the form $P = \{C, V, v\}$. An advertisement $a = \{P_1, \ldots, P_n\}$ is said to be a match for a query $q = \{P_1, \ldots P_m\}$ if and only if $q \subseteq a$.

There are several different approaches to distributing the indexing of structured documents [13][15][75][47]. The properties of the service parameter triples lend themselves well to two types of hash-function-based string indexing: string hashing (where a hash index entry is calculated from the URI representation of the entire triple) and Bloom filters (where multiple hash functions are used to create a filter from the entire set for testing the existence of individual triples in a set). Table 7.1 summarizes the reasons why each one of these techniques is suitable for indexing these service descriptions.

It is important to note that neither of these techniques is sufficient to index service descriptions effectively. While plain hash indexing is sufficient for indexing single parameter triples, it is insufficient for indexing entire service descriptions: hashing an entire set of service parameters at once allows only exact queries, not subset queries. On the other hand, using a pure Bloom filter approach forces a hierarchical tree structure on the index, making it impractical for wide area distribution in a peer-to-peer network.

We therefore choose a novel approach that combines the two indexing techniques to create a hybrid index structure. To help with the hybridization, we use an additional piece of information in the service description: the service category. A hash on the service category serves as a shortcut to narrow down advertisements and queries. This hash serves as a pointer to Bloom filters of service parameters previously advertised within this category. Each Bloom filter, in turn, points to a one or more service descriptions that it represents. Figures 7.1 and 7.2 illustrate how a single service description is turned into index entries, and Figure 7.3 shows the logical index organization of the directory.

To generate a filter of a service description, we take advantage of the additive property of Bloom filters. Each parameter triple is first converted into a unique, atomic chunk of data by concatenating the fully-qualified URIs of all of the triple components, in the canonical order. We generate n individual Bloom filters for each of the parameters $\{P_1, \ldots, P_n\}$, and combine them using bitwise-OR to create a cumulative Bloom filter, as illustrated in Algorithm 7.1. This way, the resulting cumulative filter can be use to test the membership of each of the individual parameters or any subset of the parameters in the service description.

Now that we have a Bloom filter representing the service description, we can advertise it in the service directory. To do this, we generate a hash of the fully-qualified URI of each category-layer ontology which has been used to create parameters of this service (for example, a main category ontology and one or two generic ontologies). We associate the previously-created cumulative Bloom filter with each of these category hashes, and insert the tuple into the index (see Algorithm 7.2). Similarly, querying of the index involves first, creating a cumulative Bloom filter of the query parameters, second, querying the index by using one of the category ontologies from the query and third, by returning all the advertisements associated with this category whose Bloom filter contains the query Bloom filter by set containment (see Algorithm 7.3). The reason why we create a separate advertisement for each category-layer ontology is so that the user generating a subsequent query does not need to know every category ontology that may have been used in generating the desired service descriptions. This allows a user, for example, to query services by generic attributes such as price or location, without having to specify the functional categories of the service. On the other hand, attaching the cumulative Bloom filter of the service description to each by-category advertisement allows the directory to retain all the information about a service, and to keep queries accurate.

To illustrate the indexing algorithms with an example, we create an advertisement index from the service description from Figure 4.11. First, we create Bloom filters for each of the four service parameters, and combine them to create a simple cumulative Bloom filter of the entire service description. Then, we pair this Bloom filter with a hash

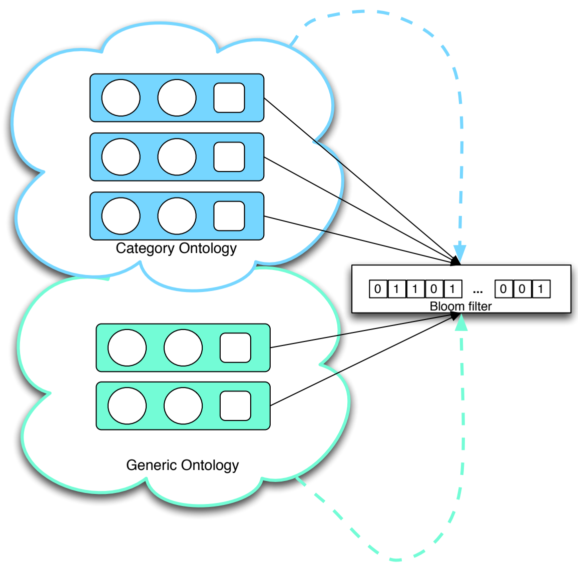


Figure 7.1: Grouping parameter triples according to their parent ontology

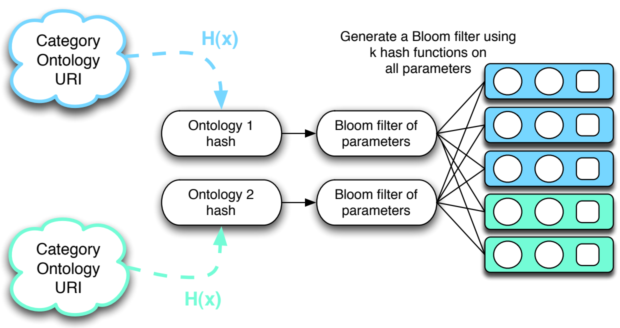

Category Ontology URI
H(x)
Generate a Bloom filter using k hash functions on all parameters
Ontology 1 hash
Bloom filter of parameters
Ontology 2 hash
Bloom filter of parameters
Category Ontology URI
H(x)

Figure 7.2: Index entries from a single service description

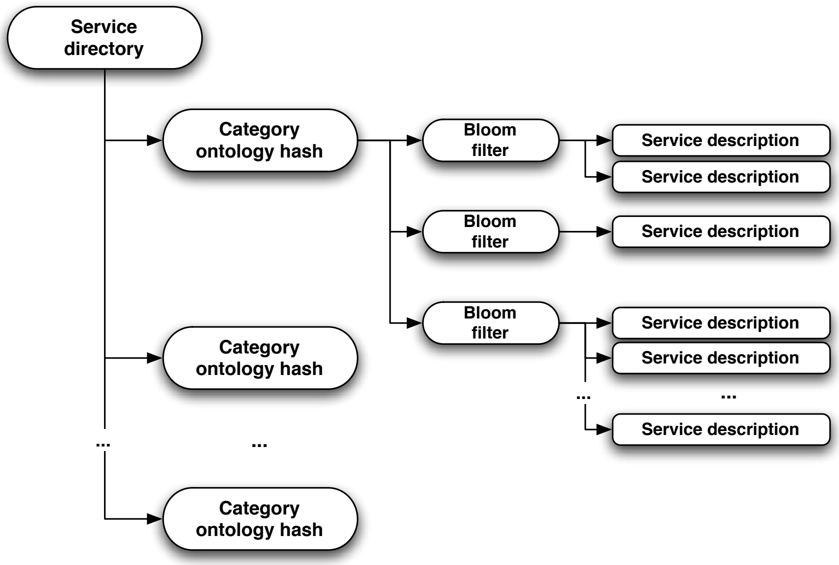


Figure 7.3: Structure of the directory index

```
Algorithm 7.1 generateFilter($s$)
Input  : Set of service parameter triples $s = \{P_1, \ldots, P_n\}$
Input  : Hash functions $\{h_1, \ldots h_k\}$ to use in the Bloom filter
Output: $Filter_s$, a Bloom filter of length $m$ containing entries for $\{P_1, \ldots, P_n\}$
for $i \in \{1, \ldots, n\}$ do
    for $j \in \{1, \ldots, k\}$ do
        Let $P_i = \{C_i, V_i, v_i\}$ be the $i$-th service parameter of $s$   Let $C_i.URI$, $V_i.URI$ and
        $v_i.URI$ be the fully-qualified URIs of $C_i$, $V_i$, and $v_i$
        Set $Filter_s[h_j((C_i.URI, V_i.URI, v_i.URI)) \mod m]$ to 1
    end
end
return $Filter_s$
```

Algorithm 7.2 advertiseIndex(a, h)

```
Input  : Advertisement $a = \{P_1, \ldots, P_n\}$ such that $P_i = \{C_i, V_i, v_i\}$ for $i \in 1 \ldots n$
Input  : Hash function $h$ used for category-layer index
$Filter_a = generateFilter(a)$
Let $\Omega$ be a set of category ontologies used in $a$   for $i \in \{1, \ldots, n\}$ do
    Let $O_i$ be the parent category ontology of $C_i$ if  $O_i \notin \Omega$ (haven't seen this category
    before) then
        Let $O_i.URI$ be the fully-qualified URI of $O_i$   Advertise $(h(O_i.URI), Filter_a)$
    end
end
```

Algorithm 7.3 queryIndex(q, h)

```
Input  : Query $q = \{P_1, \ldots, P_n\}$ such that $P_i = \{C_i, V_i, v_i\}$ for $i \in 1 \ldots n$
Input  : Hash function $h$ used for category-layer index
Output: $Results_q$, the set of advertisements matching $q$
$Filter_q = generateFilter(q)$
Let $\Omega$ be a set of category ontologies used in $q$   Let $O \in \Omega$ be a randomly-chosen category
ontology   Let $O.URI$ be the fully-qualified URI of $O$   For all advertisements $a$ stored
under the category index $h(O.URI)$, let $Results_q = \{a \mid Filter_a \supseteq Filter_q\}$ ; return
$Results_q$
```

of the URI of each of the two category-layer ontologies used in the service description: `LightpathParameters.owl` and `GeographicParameters.owl`, resulting in two index advertisements. We illustrate this indexing process in Figure 7.4.

During a subsequent query, we use the same process for the query as for an advertisement, creating a cumulative Bloom filter that represents all of the service parameters in the query. However, instead of creating a hash of each of the ontology URIs, we simply choose one at random, resulting in a single index entry. Since an identical advertisement was associated with each of the category hashes, the random choice does not affect query completeness or correctness. In the example illustrated in Figure 7.5, we hash only the full URI of `LightpathParameters.owl`. We repeat this process for each of the query synonyms found by the semantic matching algorithm; in the case of this example, we therefore generate a total of 8 index entries from the query synonyms illustrated in Figure 6.4. We can now see that the query index entry generated in Figure 7.5 matches the first advertisement index entry in Figure 7.4, since the hash values are identical, and the Bloom filter of the query is a subset of the Bloom filter in the advertisement.

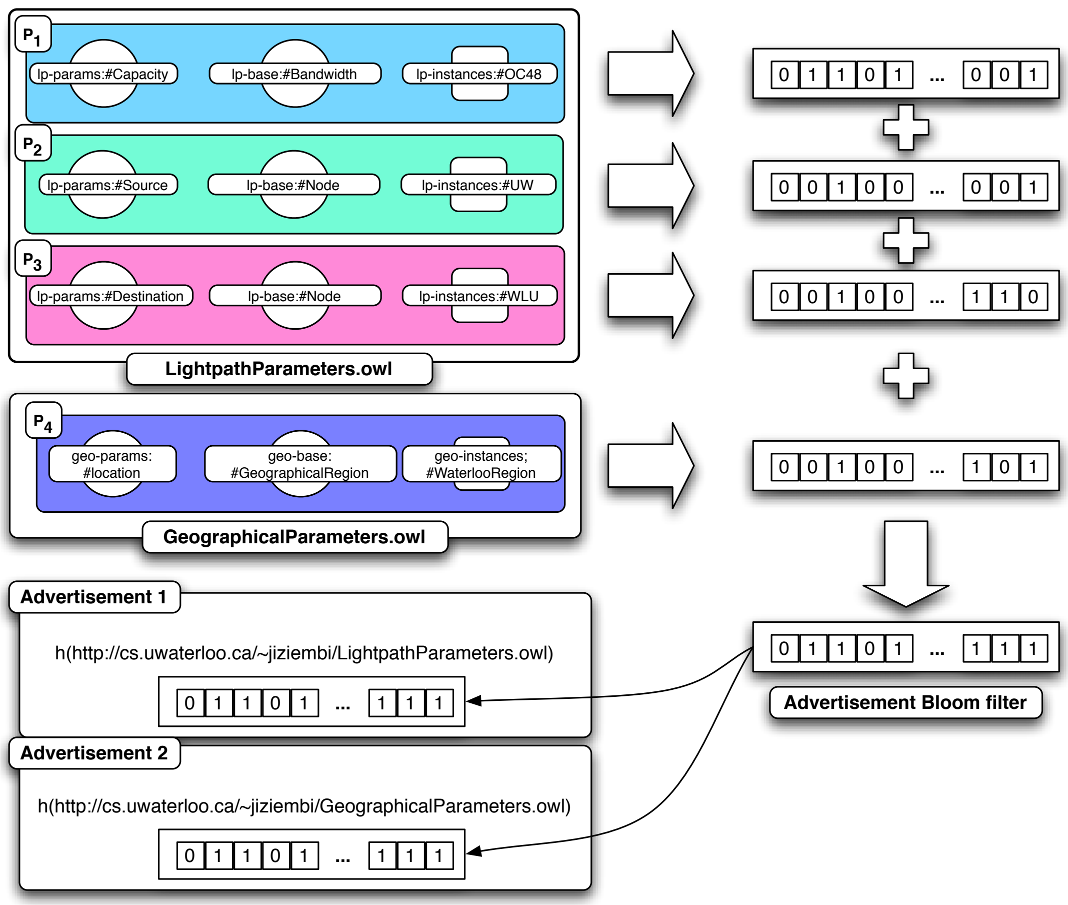


Figure 7.4: Example of creating index entries from a single service description

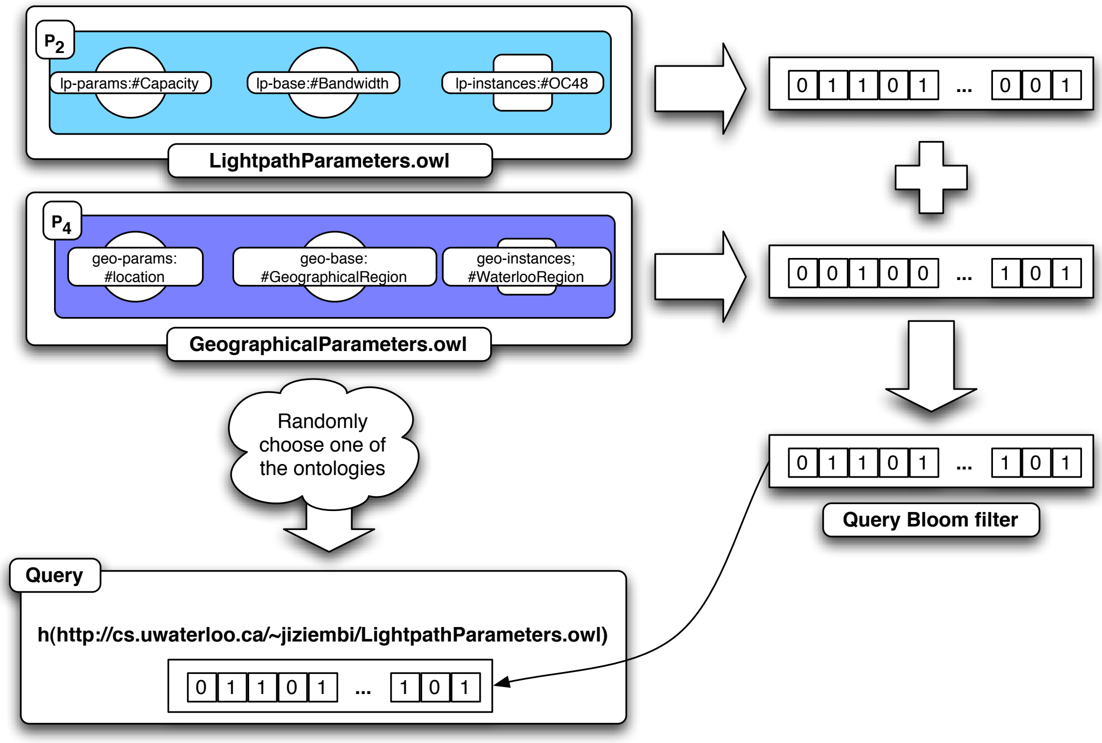


Figure 7.5: Example of creating an index entry from a single query

7.3 Data storage engine

The index described in Section 7.2 is independent of how it is physically stored; it is suitable for use in a single, central directory. However, it was designed with the specific goal of distributing it in a wide-area network. This section describes the physical layout of both the index and the data in the network overlay, in order to create a distributed index for a distributed service directory.

Because all domains must participate in the storage of the service index, it needs to be as lightweight as possible, both in terms of storage and in terms of network workload. We therefore use the principle of *leaving the intelligence at the edges*: the bulk of storage and processing of data is left to the service providers themselves, while the distributed index remains minimal. This approach serves to ensure fairness: it minimizes the amount of overhead that directory participants incur on behalf of serving index entries of participants from other administrative domains. It also makes the common elements easily deployable by directory-node operators.

We design the physical structure of the directory to mimic that of the Open Service Discovery Architecture [17]. The directory is divided into zones of control of administrative domains (in a very flexible definition of an administrative domain). The point of contact between the administrative domain and the distributed directory is called a *directory node*. This directory node allows the service providers and the service users within its home domain access to the distributed directory.
The roles of a directory node are to:

1. Provide a user interface for service users,
2. provide a user interface for service providers,
3. store and maintain the knowledge base,
4. store (or provide access to) full service descriptions from associated service providers within this domain, and
5. participate in storing the global index and in providing index information to other directory nodes in the network.

A directory node serves as a sort of gateway between the interior of an administrative domain and the services and users residing in other domains. There is, of course, no forced one-to-one correspondence between domain and directory node. There may be multiple directory nodes per domain, or one node may be shared among domains. Moreover, a directory node may not be associated with any service providers and only serve end users, or vice versa. However, to simplify the discussion, we usually assume a one-to-one domain correspondence, and a full functionality for each directory node.

7.3.1 Storing the index

The directory is divided into two parts: the local index and the global index. The local index organizes service descriptions stored locally at the directory node, while the global index is part of the global organization of all services available on the network. Figure 7.6 illustrates the way directory nodes serve as a link between local indexes and the global index.

The global index is the shared part of the directory which must be accessible by all service providers and all service users. In this framework, the global index does not belong to any particular entity, but is distributed among directory nodes. Unlike in the original design of OSDA, a directory node does not store partial XML descriptions of services from other domains; the index merely contains lightweight pointers to the directory nodes where full descriptions can be retrieved. The main distinction of this index design is the clear separation between the local and the global parts of an index, and the small proportion of overhead required by the global component of the index.

The local index is a reflection of the service description database associated with the directory node. To store the global index, we use a Chord Distributed Hash Table. Each time that a service is advertised in a local index, the directory node first updates its cumulative Bloom filter for each of the categories associated with the service. It then advertises the cumulative Bloom filters in the global index DHT, using the category hash as the key. The value corresponding to the key is a tuple containing the cumulative Bloom filter and the URI where the service provider can be contacted to retrieve the service descriptions. Figure 7.7 shows the relationship between the local and global index components.

One of the advantages of using cumulative Bloom filters at the local index is that local services need to be advertised only when the Bloom filter changes or at soft-state renewal time. Since a cumulative Bloom filter is unlikely to change upon the advertisement of a service identical or very similar to one that had been previously advertised, this reduces the network traffic incurred by advertisement of many similar services by the same provider. This is a big improvement over the original design of OSDA, where such aggregation had to be performed by custom-design policies.

7.3.2 Storing the database

While some distributed storage systems store the data along with the index in locations which depend on the indexing scheme, we choose to distribute the index throughout the network, while storing the service descriptions locally with the service provider. There are several ways of organizing service descriptions in the directory node. One could choose to make the directory node serve solely as an index; each single-service Bloom filter would be associated with a URI where the full service description could be retrieved. Another option is to treat the directory node as a database of service descriptions and store the descriptions along with the local index. Table 7.2 compares the advantages of the index-

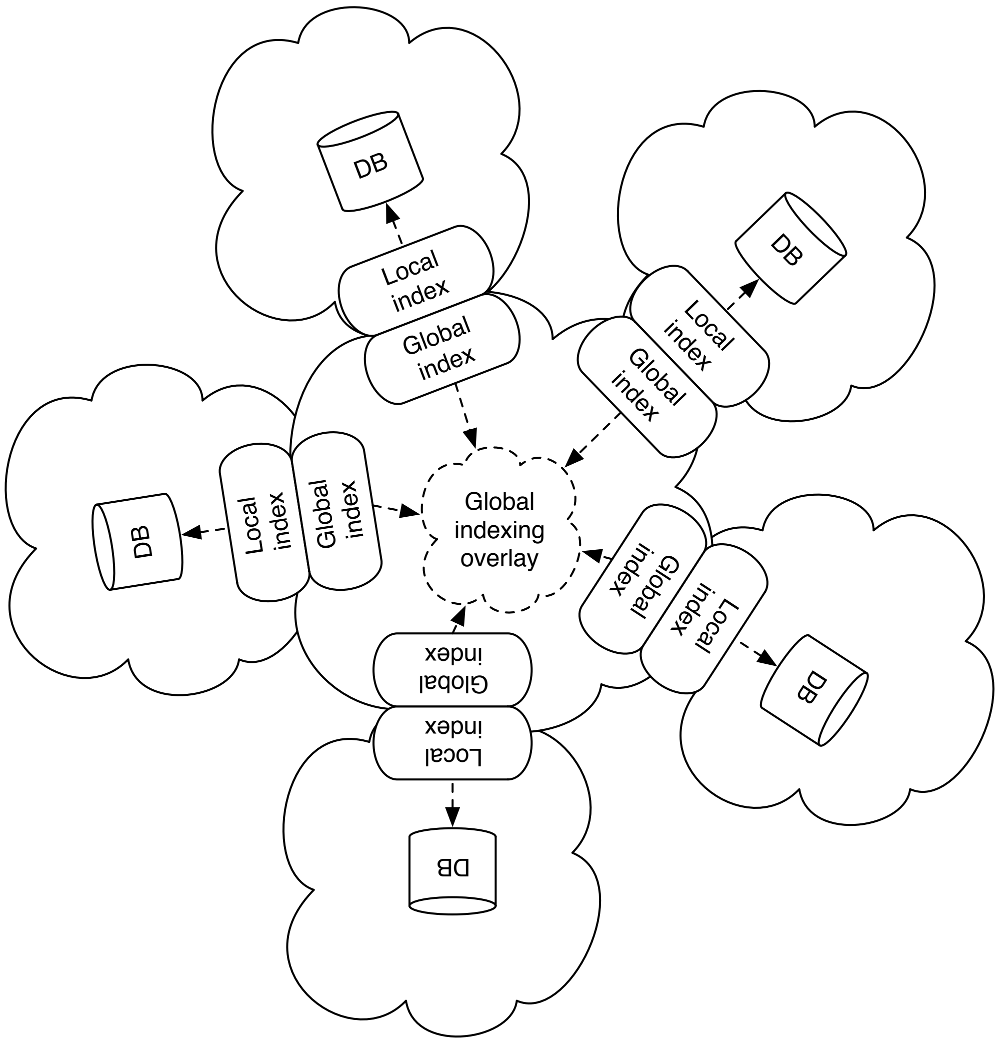


Figure 7.6: Participation of service provider domains in the global index

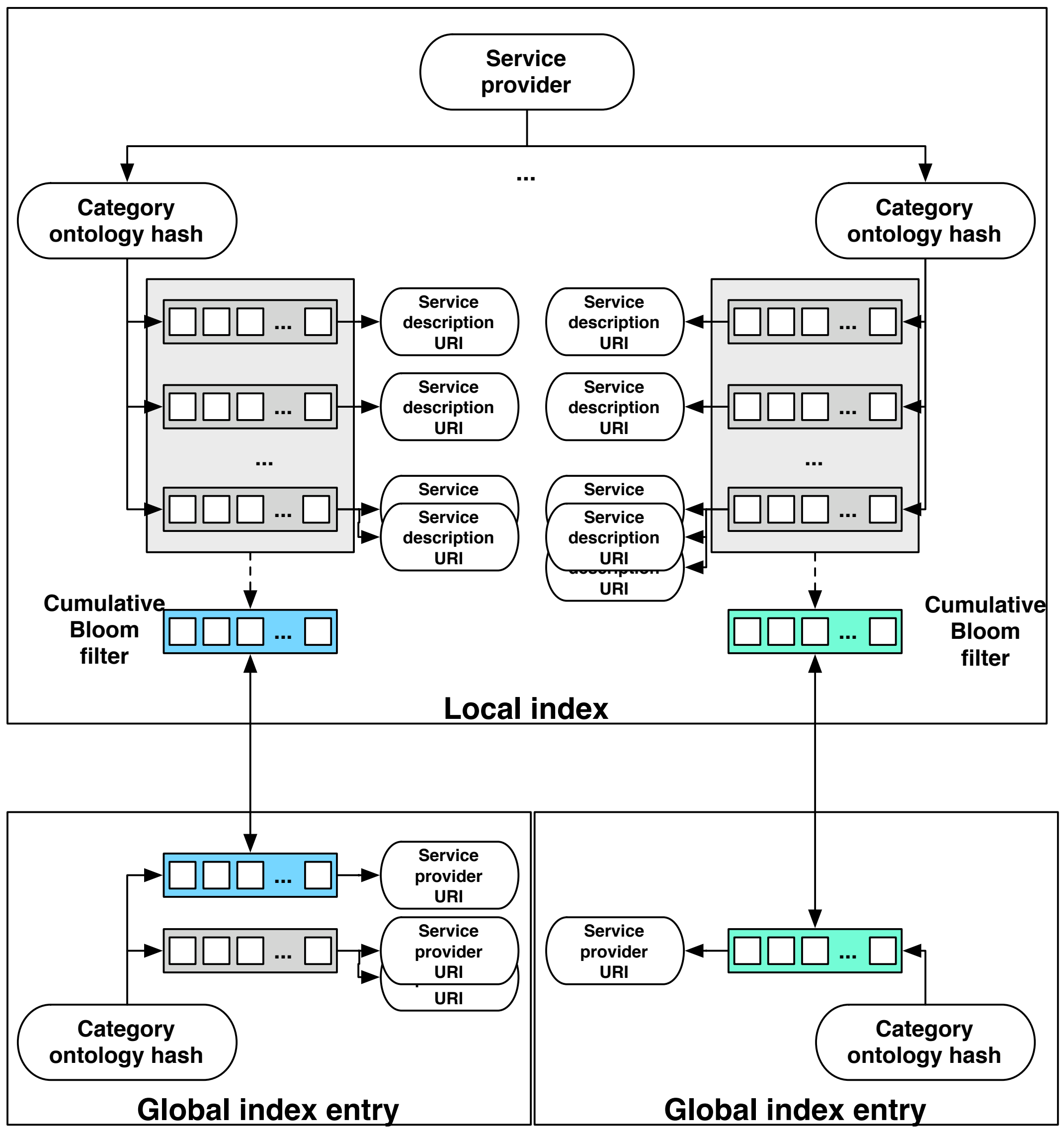


Figure 7.7: Linking the global index to the local index.

only option to those of including all indexed service descriptions at the directory node. While both approaches can easily be incorporated into the directory node, we choose to store the service description database as part of the directory node. The directory node can therefore act on behalf of a low-powered service provider (for example in a sensor network) by taking on the load of answering service queries.

Index-only directory node	More lightweight & scalable; distributes storage requirements Service provider retains independence
Database at directory node	Allows further selection of results at query-time by the directory node Allows more efficient bulk retrieval of service descriptions Can be used to implement directory-level access-control policies Service provider has fewer responsibilities

Table 7.2: Comparison of index-only directory node to storing service descriptions at the directory node (advantages of each)

7.3.3 Storing the knowledge base

One of the greatest advantages of the loosely-coupled design of the semantic search engine and indexing engine is that it does not require all directory nodes to use a single common schema. In fact, it is possible for every directory node to have a different set of ontologies. Note however, that the knowledge base determines which view the directory node posesses of the service directory as a whole. The services that can be discovered by semantic search depend wholly on the ontologies stored within the node which originated the query.

It is then obvious that for the most complete search results, each knowledge base should contain all possible ontologies used in the directory. However, retrieving and managing these ontologies poses several problems. One problem is the implementation-related task of managing and processing a potentially large knowledge base; care must be taken so that unnecessary ontologies do not take up prohibitive processing time and slow down the system. As with creating and updating ontologies, the management of a knowledge base may require human input, in order to trim it to a manageable size and relevant content.

Another problem in managing the knowledge base is discovering when new ontologies are created or existing ontologies are updated elsewhere in the service directory. This task may be performed off-line by human maintainers (not an unrealistic task, given the centralized nature of the directory node within a domain), or could be done automatically. An interesting approach to this task would be to use the existing search and indexing tools of this system to advertise new knowledge base schemas and glue ontologies as one would do for service descriptions. Glue ontologies would play a crucial role in this system, since they could be used to determine “category synonyms”, which in turn would help the directory node maintainer discover schemas describing similar types of services to the schemas already in the knowledge base. Schema distribution and discovery is hence the most important next step in improving the flexibility and usability of the framework described in this thesis.

7.4 Issues

Each design of a service directory raises its own set of problems and concerns. In this section we identify and address the issues related to the index and data distribution in this system.

7.4.1 Category-based hashing may result in a bottleneck

Using service categories as the key can create bottlenecks and unfairness in the peer-to-peer network, especially if some categories are significantly more popular than others, or if there are fewer categories than there are directory nodes. Fairness is a very important issue in making sure that the global index design is attractive to service providers from different administrative domains. Therefore, as a solution, we use several (r) hash functions to replicate each advertisement. Since advertisements are already quite inexpensive, replicating them does not incur excessive network traffic, but helps load-balance query processing and improves fairness. Subsequent queries are not replicated; instead, a single hash function is chosen randomly from the r.

7.4.2 The index does not support sophisticated queries

In contrast to many conventional databases or schema-based overlays (see Chapter 3), the global peer-to-peer index presented here does not support sophisticated query languages such as XPath, XQuery or SQL, but only a query on a subset of service parameters. In OSDA, more complex queries (such as range queries) are handled by stripping out the attributes with range directives before creating a routing key from the query and by translating the original query with XSet [40] directives[1] into an XPath query. This XPath query is then resolved at the XML database stored in each peer-to-peer node. Since this global index does not store XML service descriptions, the "piggybacked" sophisticated query must be resolved at the target service provider node in the second stage of discovery. Our approach is actually more appropriate, since the service provider stores the full service description, in contrast to the partial, aggretated service descriptions stored in OSDA peer-to-peer nodes. This allows us to handle both sophisticated queries and functional service matching in a single step during the second stage of the query. The details of this solution are not presented in this thesis, but are important future work.

[1]In XSet markup, XSetLE, XSetME, XSetLT, XSetMT denote $\leq$, $\geq$, $<$ and $>$ relationships, respectively. This allows the user to specify simple range queries.

7.4.3 Security

Any system designed for public access in the Internet may expose potentially sensitive information. Service providers may be reluctant to release the full description of their services to the public, fearing that this information could expose sensitive business information or be used to exploit security vulnerabilities. The system presented in this work has been designed to limit the amount of information exposed in the global index. The global index contains only the category hashes of services, cumulative bloom filters, and access information for the domain's second-stage discovery. Because the hashes used in the index are one-way, it is possible to discover the types of service information provided by a domain only by knowing the information in advance and using the index as a "yes-no" oracle.

In addition to restricting the information exposed in the index, keeping most of the service-description information at the service provider allows for a clean, tightly-controlled interface between the service provider and the service user. This allows the provider to implement authentication and access-control policies both for the second stage of discovery and for the service execution.

7.4.4 Each provider needs to store foreign data

In this type of index design, a directory node may need to store index data on behalf of other, possibly-competing service providers, incurring network costs in providing this data to the public. A service provider may be reluctant to incur these costs, since they do not provide any direct benefits. In a fully peer-to-peer approach, however, it is necessary to incur some costs in maintaining query-routing information.

In our design of the index, we have not eliminated the need for service providers to store data on behalf of other providers. However, we have reduced it to merely storing small cumulative index entries and the access point of a service provider. This approach is much more lightweight and predictable in terms of size when compared to other distributed indexing schemes, especially those where peers store separate chunks of information for each searchable object (e.g. a service description).

7.4.5 Excessive false positives in cumulative Bloom filters

A cumulative Bloom filter can handle only a limited number of insertions before it becomes imprecise and admits a high number of false positive results to queries. Since the probability of false positives can be calculated based on the size of the filter and the number of item insertions (See Section 2.4.1), the simple solution is to limit the number of advertisements allowed in one filter according to a set threshold of probability of false positives (for example, 1%). Then, a new cumulative filter can simply be added when the old one is "full" – the advertisement process can easily accommodate two or more cumulative category Bloom filters from the same service provider.

7.5 Complexity and overhead

7.5.1 Advertisement complexity

We measure the advertisement complexity in the distributed directory by the number of one-to-one data exchanges in the network resulting from the advertisement of a single service description (as illustrated in Figure 7.8). The first factor influencing the advertisement complexity is $|\omega|$, the number of category-layer ontologies used in the service description. Because we generate a separate hash index from the URI of each ontology in ω, $|\omega|$ is the number of separate advertisements generated from the service description. Hence, the preliminary estimate of complexity becomes:

$$AdvertisementComplexity = |\omega| \cdot \#\text{ of network accesses for each advertisement.}$$

Furthermore, suppose we have a network of n directory nodes with replication constant r, i.e. we use r hash functions, and hence, each advertisement results in r separate advertisements. Due to the properties of the Chord distributed hash table, each insert takes $O(\log(n))$ time. Therefore,

$$AdvertisementComplexity = O(r \cdot |\omega| \cdot \log n).$$

The above complexity analysis is an overestimate. In this index design, an advertisement is propagated to the network only if it results in changing the cumulative category Bloom filter at the local directory node. Recall from Section 2.4.1 that an insertion into a cumulative Bloom filter does not necessarily result in modifying the filter, and the probability of the filter remaining unmodified is equal to p, the probability of a false positive. Therefore, the probability that an advertisement will be propagated to the network is $1 - p < 1$, and

$$AdvertisementComplexity = O((1 - p) \cdot r \cdot |\omega| \cdot \log(n)).$$

In Section 2.4.1, the probability of a false positive, or idempotent insertion, in a Bloom filter is calculated assuming a uniform distribution of data. However, the data being inserted into the cumulative Bloom filter is *not* uniform, since it may consist of many similar services with many service parameters in common. If a directory node receives many identical or similar service advertisements to its local index, then p may become high, and the resulting fraction of advertisements that result in network traffic will be much reduced.

Another factor in the network traffic generated during advertisement is the size of these advertisements, which is the sum of the category hash, the advertisement Bloom filter, and the size of a URL pointing to the originating directory node. Both the URL and the hash have a negligible size of approx. 32 bytes each (if we assume MD5 is used for hashing). A high number of false positives can render the index useless, we must therefore either choose an optimal size of the Bloom filter based on a maximum number of elements it is likely to contain, or limit the numer of elements stored in a single Bloom filter. As discussed in section 7.4, a directory node can limit the number of service descriptions inserted into a single category-cumulative Bloom filter by using several filters for the same category without significantly changing the functionality of the index.

To give a rough estimate of a Bloom filter used in the directory, suppose that a cumulative Bloom filter is expected to store a maximum of 50 service descriptions from a single service provider and category. Suppose also that each service description contains at most 10 service parameters, giving a total maximum of 500 elements encoded in the Bloom filter. According to the ormula for false positives discussed in Section 2.4.1 (and summarized in a table in a technical report on this topic [76]), a ratio of 10 bits per element gives a reasonable false positive rate of approximately 0.9% with the optimal number of 5 hash functions. This results in 5000-bit (or just over half a KB) Bloom filter. If we choose to use a counting Bloom filter, the same technical report calculates that 4 bits per count are sufficient, giving generous estimate of a 2KB Bloom filter to represent five hundred service parameters. The estimate is even more generous given the fact that a service provider is likely to have many repeating service parameters in a single category, significantly reducing the true number of elements stored in the Bloom filter. The size of the Bloom filter can be further reduced by lowering its maximum capacity to less than 50 services per category.

Despite the fact that the Bloom filter of a single query has significantly fewer service parameters than a category-cumulative Bloom filter, these two filters must be the same size so that they can be compared with each other. Since there are likely to be more queries than advertisements in the global index, the size of the Bloom filter must be carefully optimized rather than simply making it arbitrarily large to avoid potential high rates of false positives.

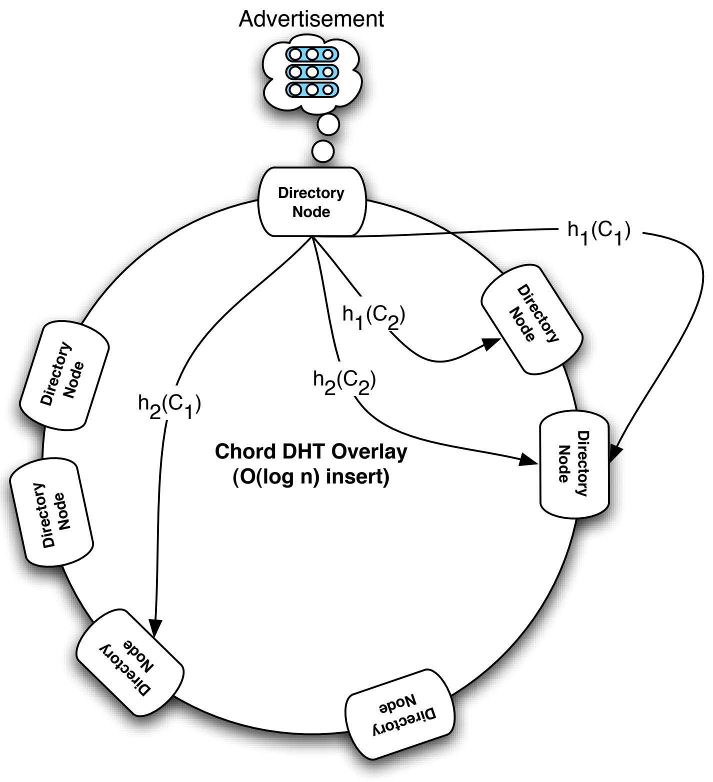
7.5.2 Query complexity

Given a query q, the key component that determines the query complexity is $|Synonyms(q)|$, analyzed in Section 6.3.5. Since each query synonym results in a single $O(\log n)$ query to the global index, the complexity of the first stage of a query is simply $O(|Synonyms(q)| \cdot \log n)$. The bounded $O(\log n)$ query complexity is determined by the properties of the Chord DHT in the overlay.

In the second stage of the query (as illustrated in Figure 7.9), the directory node which had originated the query contacts the "home" directory node of each of the K service providers who have been found in the first pass to (potentially) possess matching advertisements. Assuming $O(1)$ cost of contacting the service provider, the final complexity of the query is:

$$QueryComplexity = O(K + |Synonyms(q)| \cdot \log n).$$

While the size $|Synonyms(q)|$ of query expansion depends on the structure of the knowledge base in the directory node, the number K of service providers contacted in the



The advertisement has two categories – C_1 and C_2, and uses replication factor $r = 2$ with hash functions h_1 and h_2.

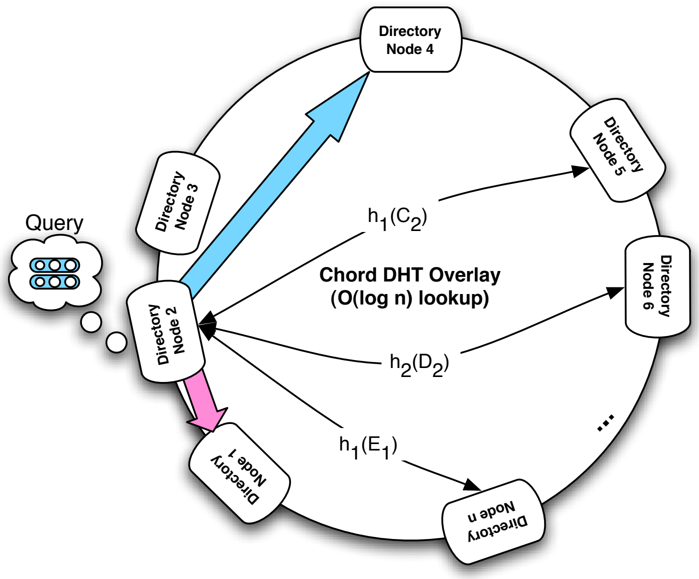
Figure 7.8: Illustration of advertisement complexity

second stage of the query is dependent on the properties of the entire global directory, i.e. how many providers have advertised matching services. Both of these factors can become pretty large, but can be reduced by the following measures.

- Cap the number of query synonyms generated from a single knowledge-domain ontology (as discussed in Section 6.3.5).
- Restrict the number of service providers contacted during the second stage of advertisement.
- Consolidate the queries sent to a single directory node (i.e., avoid sending more than one query to a directory node).

Although the number of actual number of queries and service providers may become relatively high, the second stage queries are actually independent of each other (and so are first-stage queries, if we do not consolidate service providers). Therefore, the response time of the query is as quick as the time for a single query to a service provider to be processed, with the rest of the responses arriving as they are processed. The user can therefore choose to receive only a small number of these responses, reducing the total network cost of the query, or choose to wait longer for more complete search results.

In summary, the indexing infrastructure designed in this chapter allows for very inexpensive advertisements and for queries that are likely to be executed in parallel in different parts of the network. The lightweight global index ensures scalability of the system beyond that of standard broadcast-based or DHT-based peer-to-peer networks. Because of the features of the Chord distributed hash table, and of the "no false negative" property of Bloom filters, the search process guarantees completeness of query results. Lastly, the organization of the index and the data caters to the needs of individual domains and service providers: it reduces the amount of data that a domain is obliged to store on behalf of other domains, while the two-stage query process allows directory nodes to set policies for access to full service description data.



The query has three synonyms with categories $C_1 \ldots C_{k_C}$, $D_1 \ldots C_{k_D}$ and $E_1 \ldots E_{k_E}$ respectively. The first step of query process contacts the global index component of directory nodes 5, 6 and n, while the second step contacts the local index component of directory nodes 1 and 4.

Figure 7.9: Illustration of query complexity.

Chapter 8

Implementation

The architecture presented in this thesis is being implemented using Open Service Discovery Architecture (OSDA) [17] as a basis. In fact, it fits in as a replacement for OSDA's service description and peer-to-peer component. All modules are implemented in Java and designed to use open-source, platform-independent components. Both implementation and testing are performed on Linux and Mac OS X.

In this chapter we describe the implementation of each of the components of the service directory.

8.1 Knowledge base

The knowledge base consists of a set of OWL files stored in the filesystem, and the knowledge base software module handles loading and unloading these ontology files into memory. When new ontologies are introduced, their URLs are added to a text configuration file and stored in a disk cache, or loaded from the Web. This allows ontologies to be updated frequently at the expense of some extra network traffic. Besides handling the resident set of ontologies, the knowledge base package includes the functionality of service description, querying, and advertisement in terms of OWL-S parameters.

The core of the knowledge-base is Mindswap's OWL-S API [77], which together with the Jena OWL parser [78] is used to manage sets of ontologies as a single unit, and to extract OWL-S ServiceParameters from service descriptions.

8.2 User interface engine

Service advertisements and queries are submitted using a very simple web-based implementation of the user interface. The front-end is based on OSDA's JSP web portal, while the back-end has been modified to use OWL ontologies instead of XML templates. The back-end, based on Enterprise Java Beans uses the knowledge-base module to process ontologies in order to display a hierarchical set of options.

As in OSDA, the user interface back-end runs as a web application within the Jetty[1] web server.

8.3 Semantic search engine

The semantic search engine is accessed for queries. It is not built as a separate module, but instead functions as a part of the knowledge base. The objects corresponding to service parameters and queries have the ability to build their own synonyms according to the resident knowledge base.

In addition to the Jena OWL and Mindswap OWL-S parsing libraries, the semantic search engine uses Mindswap's Pellet reasoner [79] to build parameter and query synonyms.

8.4 Indexing engine

The indexing engine takes as input a set of service parameters and creates (a) a Bloom filter representing the service and (b) a category hash. It is also used to create cumulative Bloom filters for category summaries.

We use Hongbin Liu's Java Bloom filter libraries[2]. Since we use a soft-state approach, where service advertisements expire over time, we need to use counting Bloom filters to avoid stale entries. Counting Bloom filters, however, use more space than regular Bloom filters. To preserve space, we use non-counting Bloom filters for summarizing single service descriptions within the local index, and counting Bloom filters for cumulative summaries sent to the global index.

The category hash is created using the MD5 Message Digest algorithm. Currently, the replication factor r is set at 1, that is, no hash-based replication is performed at the peer-to-peer layer.

8.5 Data storage engine

8.5.1 Storing the local index and service descriptions

The local index and full OWL-S service descriptions are stored using the open-source eXist XML database[3]. The interface with the local (to the domain) discovery system has remained similar to that in OSDA. For both local queries or advertisements, and requests in a second-stage of service discovery, interaction with the directory node's local index

[1]http://jetty.mortbay.org/jetty/
[2]http://grids.ucs.indiana.edu/ lhb/software/bloomfilter.html
[3]http://exist.sourceforge.net/

and service description store is exposed as a SOAP-based Web service running on the Apache Axis[4] platform attached to a Jetty server.

8.5.2 Storing the global index

The global index is implemented using Meteor[5], a Chord DHT implementation based on the JXTA peer-to-peer framework. The Meteor source code was modified queries consisting of tuples $(category, Bloomfilter)$ rather than key-only queries. Additionally, the Meteor $(key, value)$ tuple storage system has been modified to handle storing the global index as described in Figure 7.7. OSDA broker URIs are used as access pointers for the second stage of query.

8.6 Implementation issues

The design of the prototype implementation of this service directory is in many ways similar to the design of its architecture: it consists of several pre-existing components held together by a fair bit of "glue". The bulk of the implementation effort consists of manipulating service description and queries, and modifying the existing components to communicate with each other.

One of the most important difficulties with this design of the service directory is that of speed. The current tools for parsing Web ontologies, as well as the Meteor communication libraries are largely prototypical, and as such, they are not fast enough for a large-scale production environment. They also tend to consume a great deal of processor time and memory even when under a light load. In order for this architecture to truly function at Internet-scale, or even enterprise-scale, it needs more mature and more efficient implementations of its core components. As industry, governments and the scientific community continue to adopt Semantic Web- and peer-to-peer technogies, it is not unreasonable to hope that such implementations will eventually become available.

[4]http://ws.apache.org/axis/
[5]http://meteor.jxta.org/

Chapter 9

Conclusions and future work

This thesis describes a comprehensive framework for a Web service directory that allows user-originated, cross-domain semantic search on non-functional service parameters. As such, it introduces several features currently missing in current works in Web service description and discovery: a structured schema for describing non-functional service parameters, user-friendly semantic search of non-functional parameters and a provider-friendly cross-domain index of service descriptions.

The core of the service directory is the service description scheme, which imposes a coherent structure on OWL-S ServiceParameters and their source ontologies. Along with easier parsing and processing of the service description, this structure allows simple, template-based creation of service description and queries, benefitting both the service provider and the service user.

In addition to allowing a simple, intuitive user interface, the structure of the service description schema serves as a basis for the second main contribution of this thesis: an approach for simple mapping between service-related ontologies using "glue" ontologies, and a semantic query expansion algorithm that takes advantage of these mappings to expand the user's query. The end-user is therefore rewarded with more complete and precise search results than with keyword-search or plain attribute-value search. Meanwhile, the service provider benefits from being able to provide service descriptions using a single vocabulary and have them searchable using different vocabularies without the need to individually translate every service description.

Another main contribution of this thesis extends the search for services into large multi-domain networks by providing an innovative approach to storing and indexing service descriptions. We further take advantage of the simplicity of the service description schema by using Bloom filters combined with hash functions to create small index entries that can be searched more quickly than the text of service descriptions. The cumulative versions of these index entries are stored in a distributed hash table for cross-domain service discovery. The two-level index cleanly separates the provider's local service database (which may contain sensitive information about service access points) from the lightweight

DHT-based global index (which contains only aggregate summaries of the services offered by providers). This separation allows service providers to control access to their data, and minimizes the amount of index information that must be stored on behalf of other, possibly-competing service providers. The user, in turn, gains an ability to simultaneously search many service providers (using several different vocabularies through semantic query expansion) with only a single query.

As with all large frameworks, there are a few major difficulties in the design and implementation of this service directory. We will discuss them here in the context of potential future work that would help resolve these difficulties.

One of the difficulties is that the service description schema would require service providers to develop ontologies for different service types, and to carefully annotate the non-functional parameters of each service. The difficulty lies both in the effort required to physically compose these ontologies, and in the actual task of classifying and disambiguating service-related concepts. Once a comprehensive collection of ontologies becomes available, developing service descriptions and service-category ontologies would become much simpler, but the initial creation of such a collection would require a significant effort. Since the creation of subject-specific ontologies for all possible service types is far beyond the scope of this work, future work in this project should concentrate on ways to disseminate ontologies among service providers to avoid duplication of effort. A unified way of searching for service templates could be accomplished, for example, by introducing a category-based semantic overlay on top of the peer-to-peer system that associates service types with each other based on their real-world semantic relationships. In general, the framework would benefit from an even more systematic approach to service ontologies that would provide service providers with structured guidance in designing and expanding service templates and for creating glue ontologies.

The design of the semantic search algorithm introduces another difficulty. Depending on the complexity of the underlying ontologies, a single end-user query could generate an unmanageably-large number of synonym queries, and create a large communication load on the global index. While the algorithm itself limits the number of query synonyms by creating only those that might realistically be created by an end-user, it may be necessary to equip the search engine with a throttling mechanism that limits the number of queries sent into the network. Moreover, since the global index uses the service category as a DHT key, a popular service category could result in an uneven query load on selected directory nodes despite the small size of the index. It would be worthwhile to find more intelligent ways to balance the load on the global index, beyond the measures already presented in this thesis.

In the prototype implementation, the speed of currently-available components and libraries is proving a noticeable obstacle towards wide-area deployment of this framework. In order for the service directory to function on a truly large scale, it needs to be implemented with faster, more lightweight versions of the distributed hash table and web ontology parsers than are currently available.

Finally, it would be worthwhile to address support for more sophisticated queries in the service directory, with a good analysis of what types of queries should or should not be handled in the global index. It would be especially important to address handling dynamically-changing service attributes such as availability or current load, since these are often quite important to the end user.

.

Appendix A

Sample ontologies

A.1 Example of a service description ontology

This ontology is an OWL-S description of a lightpath service (only the `ServiceProfile` is elaborated):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE uridef [
<!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
<!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
<!ENTITY owl "http://www.w3.org/2002/07/owl">
<!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
<!ENTITY service "http://www.daml.org/services/owl-s/1.1/Service.owl">
<!ENTITY profile "http://www.daml.org/services/owl-s/1.1/Profile.owl">
<!ENTITY process "http://www.daml.org/services/owl-s/1.1/Process.owl">
<!ENTITY  grounding "http://www.daml.org/services/owl-s/1.1/Grounding.owl">
<!ENTITY lp-params "http://www.uwaterloo.ca/~jiziembi/owl/LightpathParameters.owl">
<!ENTITY lp-instances "http://www.uwaterloo.ca/~jiziembi/owl/LightpathInstances.owl">
<!ENTITY geo-params "http://www.uwaterloo.ca/~jiziembi/owl/GeographicalParameters.owl">
<!ENTITY geo-instances "http://www.uwaterloo.ca/~jiziembi/owl/GeographicalInstances.owl">
<!ENTITY DEFAULT "http://www.uwaterloo.ca/~jiziembi/owl/lp-example.owl">
]>

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdfs="&rdfs;#"
  xmlns:xsd="&xsd;#"
  xmlns:service="&service;#"
  xmlns:profile="&profile;#"
  xmlns:process="&process;#"
  xmlns:grounding="&grounding;#"
  xmlns:lp-params="&lp-params;#"
  xmlns:lp-instances="&lp-instances;#"
  xmlns:geo-params="&geo-params;#"
  xmlns:geo-instances="&geo-instances;#"
  xml:base="&DEFAULT;">

<owl:Ontology rdf:about="&DEFAULT;">
    <rdfs:comment>An example lightpath service description ontology</rdfs:comment>
    <owl:imports> <owl:Ontology rdf:about="&service;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&profile;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&process;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&grounding;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&lp-params;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&lp-instances;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&geo-params;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&geo-instances;"/> </owl:imports>
</owl:Ontology>

<!-- Service description -->
<service:Service rdf:ID="lp-example">
    <service:presents rdf:resource="#lp-exampleProfile"/>
    <service:describedBy rdf:resource="#lp-exampleProcess"/>
    <service:supports rdf:resource="#lp-exampleGrounding"/>
</service:Service>

<!-- Profile description -->
<profile:Profile rdf:ID="lp-exampleProfile">
    <service:isPresentedBy rdf:resource="#lp-example"/>

    <profile:serviceName xml:lang="en">lp-example</profile:serviceName>
    <profile:textDescription xml:lang="en">
    An example lightpath service profile, demonstrating ontology layering
    and the use of ServiceParameters
    </profile:textDescription>

   <profile:serviceParameter>
   <lp-params:Capacity rdf:ID="Capacity">
       <profile:serviceParameterName rdf:datatype="&xsd;#string">
       Capacity
       </profile:serviceParameterName>
       <profile:sParameter rdf:resource="&lp-instances;#OC48"/>
   </lp-params:Capacity>
   </profile:serviceParameter>

   <profile:serviceParameter>
   <lp-params:Source rdf:ID="Source">
          <profile:serviceParameterName rdf:datatype="&xsd;#string">
          Source
          </profile:serviceParameterName>
       <profile:sParameter rdf:resource="&lp-instances;#UW"/>
   </lp-params:Source>
   </profile:serviceParameter>
```

```
    <profile:serviceParameter>
    <lp-params:Destination rdf:ID="Destination">
        <profile:serviceParameterName rdf:datatype="&xsd;#string">
        Destination
        </profile:serviceParameterName>
        <profile:sParameter rdf:resource="&lp-instances;#WLU"/>
    </lp-params:Destination>
    </profile:serviceParameter>

    <profile:serviceParameter>
    <geo-params:Location rdf:ID="Location">
        <profile:serviceParameterName rdf:datatype="&xsd;#string">
        Location
        </profile:serviceParameterName>
        <profile:sParameter rdf:resource="&geo-instances;#Waterloo"/>
    </geo-params:Location>
    </profile:serviceParameter>

 </profile:Profile>


<!-- Process Model description -->
<process:AtomicProcess rdf:ID="lp-exampleProcess">
    <rdfs:comment>A stub process</rdfs:comment>
    <service:describes rdf:resource="#lp-example"/>
</process:AtomicProcess>

<!-- Grounding description -->
<grounding:WsdlGrounding rdf:ID="lp-exampleGrounding">
    <rdfs:comment>A stub grounding</rdfs:comment>
    <service:describes rdf:resource="#lp-example"/>
</grounding:WsdlGrounding>
```

A.2 Examples of category ontologies

A.2.1 Lightpath category ontology

This ontology contains a set of service parameters relevant to a lightpath service.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY profile "http://www.daml.org/services/owl-s/1.1/Profile.owl">
  <!ENTITY lp-base "http://www.uwaterloo.ca/~jiziembi/owl/LightpathBase.owl">
  <!ENTITY geo-base "http://www.uwaterloo.ca/~jiziembi/owl/GeographicalBase.owl">
  <!ENTITY DEFAULT "http://www.uwaterloo.ca/~jiziembi/owl/LightpathParameters.owl">
]>

<rdf:RDF
  xmlns:rdf= "&rdf;#"
  xmlns:owl="&owl;#"
  xmlns:rdfs= "&rdfs;#"
  xmlns:xsd= "&xsd;#"
  xmlns:profile= "&profile;#"
  xmlns:lp-base = "&lp-base;#"
  xmlns= "&DEFAULT;#"
  xml:base= "&DEFAULT;"
>

<owl:Ontology rdf:about="&DEFAULT;">
    <rdfs:comment>An ontology of lightpath service parameters</rdfs:comment>
    <owl:imports> <owl:Ontology rdf:about="&profile;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&lp-base;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&geo-base;"/> </owl:imports>
</owl:Ontology>

<!-- Service Parameters -->

<owl:Class rdf:ID="Capacity">
    <rdfs:subClassOf rdf:resource="&profile;#ServiceParameter"/>
    <rdfs:subClassOf>
    <owl:Restriction>
        <owl:onProperty rdf:resource="&profile;#sParameter"/>
        <owl:allValuesFrom rdf:resource="&lp-base;#Bandwidth"/>
    </owl:Restriction>
</rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Source">
<rdfs:subClassOf rdf:resource="&profile;#ServiceParameter"/>
<rdfs:subClassOf>
    <owl:Restriction>
        <owl:onProperty rdf:resource="&profile;#sParameter"/>
        <owl:allValuesFrom rdf:resource="&lp-base;#Node"/>
    </owl:Restriction>
</rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Destination">
<rdfs:subClassOf rdf:resource="&profile;#ServiceParameter"/>
<rdfs:subClassOf>
    <owl:Restriction>
        <owl:onProperty rdf:resource="&profile;#sParameter"/>
        <owl:allValuesFrom rdf:resource="&lp-base;#Node"/>
    </owl:Restriction>
```

```
</rdfs:subClassOf>
</owl:Class>
```

A.2.2 Generic category-layer ontology of geographical terms

This ontology contains a set of generic service parameters related to geographical location.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY profile "http://www.daml.org/services/owl-s/1.1/Profile.owl">
  <!ENTITY geo-base "http://www.uwaterloo.ca/~jiziembi/owl/GeographicalBase.owl">
  <!ENTITY DEFAULT "http://www.uwaterloo.ca/~jiziembi/owl/GeographicalParameters.owl">
]>

<rdf:RDF
  xmlns:rdf= "&rdf;#"
  xmlns:owl="&owl;#"
  xmlns:rdfs= "&rdfs;#"
  xmlns:xsd= "&xsd;#"
  xmlns:profile= "&profile;#"
  xmlns:geo-base = "&geo-base;#"
  xmlns= "&DEFAULT;#"
  xml:base= "&DEFAULT;"
>

<owl:Ontology rdf:about="&DEFAULT;">
    <rdfs:comment>
    A generic ontology of geographical service parameters
    </rdfs:comment>
    <owl:imports> <owl:Ontology rdf:about="&profile;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&geo-base;"/> </owl:imports>
</owl:Ontology>

<!-- Service Parameters -->

<owl:Class rdf:ID="Location">
    <rdfs:subClassOf rdf:resource="&profile;#ServiceParameter"/>
    <rdfs:subClassOf>
    <owl:Restriction>
        <owl:onProperty rdf:resource="&profile;#sParameter"/>
        <owl:allValuesFrom rdf:resource="&geo-base;#GeographicalRegion"/>
    </owl:Restriction>
</rdfs:subClassOf>
```

```
</owl:Class>
```

A.3 Examples knowledge domain ontologies of types

A.3.1 Lightpath ontology of types

This ontology contains classes relevant to a lightpath service.

```
<?xml version='1.0' encoding='ISO-8859-1'?>

<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY DEFAULT "http://www.uwaterloo.ca/~jiziembi/owl/LightpathBase.owl">
]>

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdfs="&rdfs;#"
  xmlns:xsd="&xsd;#"
  xmlns= "&DEFAULT;#"
  xml:base="&DEFAULT;"
>

<owl:Ontology rdf:about="&DEFAULT;">
    <rdfs:comment>
    A knowledge-domain ontology of lightpath service concepts
    </rdfs:comment>
</owl:Ontology>

<owl:Class rdf:ID="Bandwidth">
    <rdfs:comment>A class of lightpath bandwidth values</rdfs:comment>
</owl:Class>

<owl:Class rdf:ID="Node">
    <rdfs:comment>A class of lightpath endpoints</rdfs:comment>
</owl:Class>

<owl:DatatypeProperty rdf:ID="NodeName">
    <rdfs:domain rdf:resource="#Node"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="NodeIP">
    <rdfs:domain rdf:resource="#Node"/>
```

```
</owl:DatatypeProperty>
```

A.3.2 Generic ontology of types

This generic ontology contains classes of relevant to a lightpath service.

```
<?xml version='1.0' encoding='ISO-8859-1'?>

<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY DEFAULT "http://www.uwaterloo.ca/~jiziembi/owl/GeographicalBase.owl">
]>

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdfs="&rdfs;#"
  xmlns:xsd="&xsd;#"
  xmlns=      "&DEFAULT;#"
  xml:base="&DEFAULT;"
>


<owl:Ontology rdf:about="&DEFAULT;">
    <rdfs:comment>
    A small knowledge-domain ontology of geographical concepts
    </rdfs:comment>
</owl:Ontology>

<!-- Geographical regions -->
<owl:Class rdf:ID="GeographicalRegion">
    <rdfs:comment>A generic class of geographical region</rdfs:comment>
</owl:Class>

<owl:Class rdf:ID="City">
    <rdfs:subClassOf rdf:resource="#GeographicalRegion"/>
</owl:Class>

<owl:Class rdf:ID="County">
    <rdfs:subClassOf rdf:resource="#GeographicalRegion"/>
</owl:Class>

<owl:Class rdf:ID="Province">
    <rdfs:subClassOf rdf:resource="#GeographicalRegion"/>
</owl:Class>
```

```
<owl:Class rdf:ID="Country">
    <rdfs:subClassOf rdf:resource="#GeographicalRegion"/>
</owl:Class>

<!-- Relaxing and subsuming properties -->
<owl:TransitiveProperty rdf:ID="locatedIn">
    <rdfs:domain rdf:resource="#GeographicalRegion"/>
    <rdfs:range rdf:resource="#GeographicalRegion"/>
</owl:TransitiveProperty>

<owl:TransitiveProperty rdf:ID="contains">
    <owl:inverseOf rdf:resource="#locatedIn" />
    <rdfs:domain rdf:resource="#GeographicalRegion"/>
    <rdfs:range rdf:resource="#GeographicalRegion"/>
</owl:TransitiveProperty>
```

A.4 Example of a knowledge domain ontology of instances

A.4.1 Lightpath ontology of types

This ontology contains instances of lightpath-related objects.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY lp-base "http://www.uwaterloo.ca/~jiziembi/owl/LightpathBase.owl">
  <!ENTITY DEFAULT "http://www.uwaterloo.ca/~jiziembi/owl/LightpathInstances.owl">
]>

<rdf:RDF
  xmlns:rdf= "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:owl= "http://www.w3.org/2002/07/owl#"
  xmlns:rdfs= "&rdfs;#"
  xmlns:xsd= "&xsd;#"
  xmlns:lp-base= "&lp-base;#"
  xmlns=     "&DEFAULT;#"
  xml:base= "&DEFAULT;"
>

<owl:Ontology rdf:about="&DEFAULT;">
    <rdfs:comment>
    A knowledge-domain ontology of instances of  the LightpathBase concepts
    </rdfs:comment>
    <owl:imports> <owl:Ontology rdf:about="&lp-base;"/> </owl:imports>
</owl:Ontology>


<!--  Endpoint nodes -->

<lp-base:Node rdf:ID="UW">
    <lp-base:NodeName  rdf:parseType='Literal'>University of Waterloo</lp-base:NodeName>
       <lp-base:NodeIP>192.168.0.1</lp-base:NodeIP>
</lp-base:Node>

<lp-base:Node rdf:ID="WLU">
    <lp-base:NodeName>Wilfrid Laurier University</lp-base:NodeName>
       <lp-base:NodeIP>192.168.1.1</lp-base:NodeIP>
</lp-base:Node>

<lp-base:Node rdf:ID="UWO">
    <lp-base:NodeName>University of Western Ontario</lp-base:NodeName>
    <lp-base:NodeIP>192.168.2.1</lp-base:NodeIP>
</lp-base:Node>


<!-- Pre-set bandwidth values -->

<lp-base:Bandwidth rdf:ID="OC3"/>
<lp-base:Bandwidth rdf:ID="OC12"/>
<lp-base:Bandwidth rdf:ID="OC24"/>
<lp-base:Bandwidth rdf:ID="OC36"/>
<lp-base:Bandwidth rdf:ID="OC48"/>
<lp-base:Bandwidth rdf:ID="OC96"/>
<lp-base:Bandwidth rdf:ID="OC192"/>
```

A.4.2 Generic ontology of types

This generic ontology contains instances of geographical regions.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY geo-base "http://www.uwaterloo.ca/~jiziembi/owl/GeographicalBase.owl">
  <!ENTITY DEFAULT "http://www.uwaterloo.ca/~jiziembi/owl/GeographicalInstances.owl">
]>

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdfs="&rdfs;#"
  xmlns:xsd="&xsd;#"
  xmlns:geo-base="&geo-base;#"
  xmlns=      "&DEFAULT;#"
  xml:base="&DEFAULT;"
>

<owl:Ontology rdf:about="&DEFAULT;">
    <rdfs:comment>
    A knowledge-domain ontology of instances of geographical concepts from
     GeographicalBase
    </rdfs:comment>
    <owl:imports> <owl:Ontology rdf:about="&geo-base;"/> </owl:imports>
</owl:Ontology>

<geo-base:Country rdf:ID="Canada"></geo-base:Country>

<geo-base:Province rdf:ID="Ontario">
    <geo-base:locatedIn rdf:resource="#Canada" />
</geo-base:Province>

<geo-base:County rdf:ID="WaterlooRegion">
    <geo-base:locatedIn rdf:resource="#Ontario" />
</geo-base:County>

<geo-base:City rdf:ID="Waterloo">
    <geo-base:locatedIn rdf:resource="#WaterlooRegion" />
</geo-base:City>

<geo-base:City rdf:ID="Kitchener">
    <geo-base:locatedIn rdf:resource="#WaterlooRegion" />
</geo-base:City>

<geo-base:City rdf:ID="Cambridge">
    <geo-base:locatedIn rdf:resource="#WaterlooRegion" />
</geo-base:City>

<geo-base:City rdf:ID="London">
    <geo-base:locatedIn rdf:resource="#Ontario" />
</geo-base:City>
```

A.5 Examples of glue ontologies

A.5.1 Defining the synonym-generating properties

In this ontology, we define two transitive generic properties *subsumes* and *relaxes*. In order to designate properties previously-defined in knowledge-domain ontologies as subsuming or relaxing, we mark these properties as subproperties of *subsumes* or *relaxes*

```
<?xml version='1.0' encoding='ISO-8859-1'?>

<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY DEFAULT "http://www.uwaterloo.ca/~jiziembi/owl/GlueProperties.owl">
]>

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdfs="&rdfs;#"
  xmlns:xsd="&xsd;#"
  xmlns=      "&DEFAULT;#"
  xml:base="&DEFAULT;"
  >

<owl:Ontology rdf:about="&DEFAULT;">
    <rdfs:comment>
    An ontology that defines the types of properties used in glue ontologies
    </rdfs:comment>
</owl:Ontology>


<owl:TransitiveProperty rdf:ID="subsumes">
    <rdfs:domain rdf:resource="&owl;#Thing"/>
    <rdfs:range rdf:resource="&owl;#Thing"/>
</owl:TransitiveProperty>

<owl:TransitiveProperty rdf:ID="relaxes">
    <rdfs:domain rdf:resource="&owl;#Thing"/>
    <rdfs:range rdf:resource="&owl;#Thing"/>
    <owl:inverseOf rdf:resource="#subsumes"/>
</owl:TransitiveProperty>

<!-- a temporary measure to make the reasoner happy (kb needs to know about sameAs) -->
<owl:Thing rdf:ID="instance1"/>
<owl:Thing rdf:ID="instance2"/>
<rdf:Description rdf:about="#instance1">
    <owl:sameAs rdf:resource="#instance2" />
</rdf:Description>
```

A.5.2 Gluing together two ontologies

This glue ontology demonstrates the use of OWL's subclassing and equivalence vocabulary to "glue together" the classes and instances of two different category-layer ontologies. We also show how properties such as "locatedIn" and "contains" can be marked as subsuming or relaxing.

```
<?xml version='1.0' encoding='ISO-8859-1'?>

<!DOCTYPE uridef[
<!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
<!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
<!ENTITY owl "http://www.w3.org/2002/07/owl">
<!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
<!ENTITY lp-params "http://www.uwaterloo.ca/~jiziembi/owl/LightpathParameters.owl">
<!ENTITY net-params "http://www.uwaterloo.ca/~jiziembi/owl/NetworkLinkParameters.owl">
<!ENTITY geo-params "http://www.uwaterloo.ca/~jiziembi/owl/GeographicalParameters.owl">
<!ENTITY lp-base "http://www.uwaterloo.ca/~jiziembi/owl/LightpathBase.owl">
<!ENTITY net-base "http://www.uwaterloo.ca/~jiziembi/owl/NetworkLinkBase.owl">
<!ENTITY geo-base "http://www.uwaterloo.ca/~jiziembi/owl/GeographicalBase.owl">
<!ENTITY lp-instances "http://www.uwaterloo.ca/~jiziembi/owl/LightpathInstances.owl">
<!ENTITY net-instances "http://www.uwaterloo.ca/~jiziembi/owl/NetworkLinkInstances.owl">
<!ENTITY geo-instances "http://www.uwaterloo.ca/~jiziembi/owl/GeographicalInstances.owl">
<!ENTITY glue-properties "http://www.uwaterloo.ca/~jiziembi/owl/GlueProperties.owl">
<!ENTITY DEFAULT "http://www.uwaterloo.ca/~jiziembi/owl/Glue.owl">
]>

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdfs="&rdfs;#"
  xmlns:xsd="&xsd;#"
  xmlns:net-params="&net-params;#"
  xmlns:net-base="&net-base;#"
  xmlns:net-instances="&net-instances;#"
  xmlns:lp-params="&lp-params;#"
  xmlns:lp-base="&lp-base;#"
  xmlns:lp-instances="&lp-instances;#"
  xmlns:geo-params="&geo-params;#"
  xmlns:geo-base="&geo-base;#"
  xmlns:geo-instances="&geo-instances;#"
  xmlns:glue-properties="&glue-properties;#"
  xmlns=      "&DEFAULT;#"
  xml:base="&DEFAULT;"
>

<owl:Ontology rdf:about="&DEFAULT;">
    <rdfs:comment>
    A glue ontology translating between LightpathParameters.owl and
     NetworkLinkParameters.owl
     </rdfs:comment>
    <owl:imports> <owl:Ontology rdf:about="&lp-params;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&net-params;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&geo-params;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&lp-base;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&net-base;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&geo-base;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&lp-instances;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&net-instances;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&geo-instances;"/> </owl:imports>
    <owl:imports> <owl:Ontology rdf:about="&glue-properties;"/> </owl:imports>
</owl:Ontology>


 <!-- Translate at the level of parameter classes -->
<rdf:Description rdf:about="&lp-params;#Capacity">
      <rdfs:subClassOf rdf:resource="&net-params;#Capacity" />
</rdf:Description>

<rdf:Description rdf:about="&lp-params;#Destination">
      <owl:equivalentClass rdf:resource="&net-params;#Dest" />
</rdf:Description>

<rdf:Description rdf:about="&lp-params;#Source">
      <owl:equivalentClass rdf:resource="&net-params;#Src" />
</rdf:Description>


<!-- Translate at the level of instances -->

<rdf:Description rdf:about="&net-instances;#UW">
    <owl:sameAs rdf:resource="&lp-instances;#UW" />
</rdf:Description>

<rdf:Description rdf:about="&net-instances;#WLU">
    <owl:sameAs rdf:resource="&lp-instances;#WLU" />
</rdf:Description>

<rdf:Description rdf:about="&net-instances;#UWO">
    <owl:sameAs rdf:resource="&lp-instances;#UWO" />
</rdf:Description>


<rdf:Description rdf:about="&lp-instances;#OC3">
    <owl:sameAs rdf:resource="&net-instances;#STS-3"/>
</rdf:Description>

<rdf:Description rdf:about="&lp-instances;#OC12">
    <owl:sameAs rdf:resource="&net-instances;#STS-12"/>
</rdf:Description>
```

```
<rdf:Description rdf:about="&lp-instances;#OC24">
    <owl:sameAs rdf:resource="&net-instances;#STS-24"/>
</rdf:Description>

<rdf:Description rdf:about="&lp-instances;#OC48">
    <owl:sameAs rdf:resource="&net-instances;#STS-48"/>
</rdf:Description>

<rdf:Description rdf:about="&lp-instances;#OC192">
    <owl:sameAs rdf:resource="&net-instances;#STS-192"/>
</rdf:Description>


<!-- Identify the subsuming/relaxing properties: "locatedIn" and "contains" -->

<rdf:Description rdf:about="&geo-base;#locatedIn">
      <rdfs:subPropertyOf rdf:resource="&glue-properties;#subsumes"/>
</rdf:Description>

<rdf:Description rdf:about="&geo-base;#contains">
      <rdfs:subPropertyOf rdf:resource="&glue-properties;#relaxes"/>
</rdf:Description>
```

Bibliography

[1] V. Vinge, "Fast times at Fairmont High," in *The Collected Stories of Vernor Vinge*. New York, NY, USA: Tor Books, 2001.

[2] UPnP Forum, "UPnP device architecture 1.0," May 2003. [Online]. Available: http://www.upnp.org/download/ UPnPDA10_20000613.htm

[3] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan, "RFC2165: Service Location Protocol," June 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2165.txt

[4] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications.* Springer-Verlag Berlin Heidelberg, 2004.

[5] W. F. . H.-P. David Booth and S. L. Canyang Kevin Liu, "Web Service Definition Language." [Online]. Available: http://www.w3.org/TR/wsdl

[6] S. Luc Clement, I. Andrew Hately, S. A. Claus von Riegen, and C. A. Tony Rogers, "UDDI version 3.0.2," Oct. 2004, UDDI Spec Technical Committee Draft. [Online]. Available: http://uddi.org/pubs/uddi_v3.htm

[7] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara, "OWL-S: Semantic markup for web services (technical overview)." [Online]. Available: http://www.daml.org/services/owl-s/1.1/overview/

[8] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[9] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001. [Online]. Available: http://www.pdos.lcs.mit.edu/chord/papers/paper-ton.pdf

[10] H. Ishikawa, K. Kubota, and Y. Kanemasa, *XQL: A Query Language for XML Data*, Fujitsu Laboratories Ltd. [Online]. Available: http://www.w3.org/TandS/QL/QL98/pp/flab.txt

[11] Wikipedia, "SQL (Structured Query Language," Oct. 2005. [Online]. Available: http://en.wikipedia.org/wiki/SQL

[12] World Wide Web Consortium, *XQuery*, World Wide Web Consortium, 2004. [Online]. Available: http://www.w3.org/XML/Query

[13] M. Balazinska, H. Balakrishnan, and D. Karger, "INS/Twine: A scalable peer-to-peer architecture for intentional resource discovery," in *Proceedings of the First International Conference on Pervasive Computing.* Springer-Verlag, 2002, pp. 195–210.

[14] T. D. Hodes, S. E. Czerwinski, B. Y. Zhao, A. D. Joseph, and R. H. Katz, "An architecture for secure wide-area service discovery," *Wirel. Netw.*, vol. 8, no. 2/3, pp. 213–230, 2002.

[15] G. Koloniari and E. Pitoura, "Filters for XML-based service discovery in pervasive computing," *The Computer Journal*, vol. 47, no. 4, pp. 461–474, July 2004.

[16] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of ACM ASPLOS.* ACM, November 2000.

[17] N. Limam, J. Ziembicki, R. Ahmed, Y. Iraqi, D. T. Li, R. Boutaba, and F. Cuervo, "OSDA: Open service discovery architecture for efficient cross-domain service provisioning," *Computer Communications Journal, special issue for Emerging Middleware for Next Generation Networks*, 2005, to appear. [Online]. Available: http://bcr2.uwaterloo.ca/˜alcatel/publications.htm

[18] H. Haas and A. Brown, *Web Services Glossary*, World Wide Web Consortium, August 2003. [Online]. Available: http://www.w3.org/TR/2003/WD-ws-gloss-20030808/

[19] "SOAP Version 1.2 part 0: Primer," June 2003. [Online]. Available: http://www.w3.org/TR/2003/REC-soap12-part0-20030624/

[20] O. Nickolas Kavantzas, C. O. David Burdett, N. Gregory Ritzinger, C. Tony Fletcher, and W. Yves Lafon, *Web Services Choreography Description Language Version 1.0*, World Wide Web Consortium, Dec. 2004, W3C Working Draft. [Online]. Available: http://www.w3.org/TR/ws-cdl-10/

[21] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, *Web Service Architecture*, World Wide Web Consortium, February 2004. [Online]. Available: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

[22] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Service Definition Language version 2.0 part 0: Primer." [Online]. Available: http://www.w3.org/TR/2005/WD-wsdl20-primer-20050803/

[23] Internet Assigned Numbers Authority, "Uniform resource identifier (URI) schemes." [Online]. Available: http://www.iana.org/assignments/uri-schemes

[24] E. Miller, R. Swick, D. Brickley, B. McBride, J. Hendler, G. Schreiber, and D. Connolly, *Semantic Web*, World Wide Web Consortium. [Online]. Available: http://www.w3.org/2001/sw/

[25] D. Beckett, *RDF/XML Syntax Specification (Revised)*, World Wide Web Consortium, Feb. 2004. [Online]. Available: http://www.w3.org/TR/rdf-syntax-grammar/

[26] J. Handler, *Frequently Asked Questions on W3C's Web Ontology Language (OWL)*, W3C Web Ontology Working Group, Feb. 2004. [Online]. Available: www.w3.org/2003/08/owlfaq

[27] M. K. Smith, C. Welty, and D. L. McGuinness, *OWL Web Ontology Language Guide*, W3C Web Ontology Working Group, Feb. 2004. [Online]. Available: http://www.w3.org/TR/2004/REC-owl-guide-20040210

[28] I. Horrocks, F. van Harmelen, P. Patel-Schneider, T. Berners-Lee, D. Brickley, D. Connolly, M. Dean, S. Decker, D. Fensel, R. Fikes, P. Hayes, J. Heflin, J. Hendler, O. Lassila, D. McGuinness, and L. A. Stein, "DAML+OIL," Mar. 2001. [Online]. Available: http://www.daml.org/2001/03/daml+oil-index.html

[29] "The OWL-S Matchmaker." [Online]. Available: http://www-2.cs.cmu.edu/˜softagents/daml_Mmaker/daml-s_matchmaker.htm

[30] P. W. Lord, S. Bechhofer, M. D. Wilkinson, G. Schiltz, D. Gessler, D. Hull, C. A. Goble, and L. Stein, "Applying semantic web services to bioinformatics: Experiences gained, lessons learnt." in *International Semantic Web Conference*, 2004, pp. 350–364.

[31] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara, "Semantic matching of web services capabilities," in *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*. Springer-Verlag, 2002, pp. 333–347.

[32] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area Web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000. [Online]. Available: http://www.cs.wisc.edu/˜cao/papers/summarycache.ps

[33] Wikipedia, “Distributed hash table,” Sept. 2005. [Online]. Available: http://en.wikipedia.org/wiki/Distributed_hash_table

[34] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A Scalable Content Addressable Network,” in *Proceedings of ACM SIGCOMM 2001*, 2001. [Online]. Available: http://www.acm.org/sigs/sigcomm/sigcomm2001/p13-ratnasamy.pdf

[35] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” *Lecture Notes in Computer Science*, vol. 2218, pp. 329–350, 2001. [Online]. Available: http://research.microsoft.com/˜antr/PAST/pastry.pdf

[36] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, “Tapestry: An infrastructure for fault-tolerant wide-area location and routing,” UC Berkeley, Tech. Rep. UCB/CSD-01-1141, 2001. [Online]. Available: citeseer.nj.nec.com/zhao01tapestry.html

[37] P. Maymounkov and D. Mazieres, “Kademlia: A peer-to-peer information system based on the xor metric,” in *Proceedings of IPTPS02*, Mar. 2002.

[38] Caldera Systems, “OpenSLP.” [Online]. Available: http://www.openslp.org/

[39] Sun Microsystems, “Jini technology architectural overview,” Sun Microsystems, Inc, Tech. Rep., 1999. [Online]. Available: http://wwws.sun.com/software/jini/whitepapers/architecture.pdf

[40] B. Zhao, “The Xset XML search engine and XBench XML query benchmark,” University of California, Berkeley, Tech. Rep. UCB/CSD-00-1112, 2000. [Online]. Available: http://www.cs.berkeley.edu/˜ravenben/xset/

[41] K. Risvik and R. Michelsen, “Search engines and web dynamics,” 2002. [Online]. Available: citeseer.ist.psu.edu/risvik02search.html

[42] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, Inc., 1999.

[43] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica, “Complex queries in dht-based peer-to-peer networks,” in *Revised Papers from the First International Workshop on Peer-to-Peer Systems.* Springer-Verlag, 2002, pp. 242–259.

[44] R. Huebsch and B. T. Loo, “The PIER relational query processing system,” U.C. Berkeley CS Department, Tech. Rep., Feb. 2002.

[45] C. Tang and S. Dwarkadas, “Hybrid global-local indexing for efficient peer-to-peer information retrieval,” in *NSDI ’04 Proceedings of the first symposium on networked systems design and implementation*, Mar. 2004.

[46] C. Tang, Z. Xu, and S. Dwarkadas, “Peer-to-peer information retrieval using self-organizing semantic overlay networks,” in *SIGCOMM ’03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications.* New York, NY, USA: ACM Press, 2003, pp. 175–186.

[47] M. Cai, M. Frank, B. Yan, and R. M. MacGregor, “A subscribable peer-to-peer rdf repository for distributed metadata management.” *J. Web Sem.*, vol. 2, no. 2, pp. 109–130, 2004.

[48] M. Cai, M. Frank, J. Chen, and P. Szekely, “Maan: A multi-attribute addressable network for grid information services,” in *GRID ’03: Proceedings of the Fourth International Workshop on Grid Computing.* IEEE Computer Society, 2003, p. 184.

[49] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch, “EDUTELLA: a P2P networking infrastructure based on RDF,” in *WWW ’02: Proceedings of the 11th international conference on World Wide Web.* New York, NY, USA: ACM Press, 2002, pp. 604–615.

[50] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. T. Schlosser, I. Brunkhorst, and A. Löser, “Super-peer-based routing strategies for RDF-based peer-to-peer networks,” *J. Web Sem.*, vol. 1, no. 2, pp. 177–186, 2004.

[51] J. Broekstra, M. Ehrig, P. Haase, F. van Harmelen, A. Kampman, M. Saboul, R. Siebes, S. Staab, H. Stuckenschmidt, and C. Tempich, “A metadata model for semantics-based peer-to-peer systems,” in *Proceedings of the 1st Workshop on Semantics in Peer-to-Peer and Grid Computing at the Twelfth International World Wide Web Conference*, May 2003.

[52] J. Broekstra, A. Kampman, and F. van Harmelen, “Sesame: A generic architecture for storing and querying rdf and rdf schema,” in *ISWC ’02: Proceedings of the First International Semantic Web Conference on The Semantic Web.* London, UK: Springer-Verlag, 2002, pp. 54–68.

[53] J. Broekstra, “SeRQL: Sesame RDF query language.” Tech. Rep., 2003. [Online]. Available: http://swap.semanticweb.org.proxy.lib.uwaterloo.ca/public/Publications/swap-d3.2.pdf

[54] P. Haase, J. Broekstra, M. Ehrig, M. Menken, P. Mika, M. Plechawski, P. Pyszlak, B. Schnizler, R. Siebes, S. Staab, and C. Tempich, “Bibster - a semantics-based bibliographic peer-to-peer system,” in *Proceedings of the Third International Semantic Web Conference*, ser. LNCS, S. A. McIlraith, D. Plexousakis, and F. van Harmelen, Eds. Springer, Nov. 2004, pp. 122–136.

[55] C. Tempich, S. Staab, and A. Wranik, "Remindin': semantic query routing in peer-to-peer networks based on social metaphors," in *Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 640 – 649.

[56] C. Skouteli, G. Samaras, and E. Pitoura, "Concept-based discovery of mobile services," in *MEM '05: Proceedings of the 6th international conference on Mobile data management*. New York, NY, USA: ACM Press, 2005, pp. 257–261.

[57] Princeton University Cognitive Science Directory, "Wordnet." [Online]. Available: http://wordnet.princeton.edu/w3wn.html

[58] S. Castano, A. Ferrara, S. Montanelli, E. Pagain, and G. Rossi, "Ontology-addressable contents in P2P networks," in *Proc. of the 1st WWW Int. Workshop on Semantics in Peer-to-Peer and Grid Computing (SemPGRID 2003)*, May 2003.

[59] K. Arabshian and H. Schulzrinne, "GloServ: global service discovery architecture," in *The First Annual International Conference on Mobile and Ubiquitous Systems (MOBIQUITOUS): Networking and Services*, Aug. 2004, pp. 319 – 325. [Online]. Available: http://www1.cs.columbia.edu/~knarig/gloserv.pdf

[60] K. Arabshian, H. Schulzrinne, D. Trossen, and D. Pavel, "GloServ: Global service discovery using the OWL web ontology language," in *IEE International Workshop on Intelligent Environments(IE05)*. IEE, June 2005. [Online]. Available: http://www1.cs.columbia.edu/~knarig/gloservRevised.pdf

[61] K. Arabshian and H. Schulzrinne, "Hybrid hierarchical and peer-to-peer ontology-based global service discovery system," in *Columbia University Technical Report CUCS-016-05*, Apr. 2005. [Online]. Available: http://www1.cs.columbia.edu/~library/TR-repository/reports/reports-2005/cucs-016-05.pdf

[62] N. Srinivasan, M. Paolucci, and K. Sycara, "An efficient algorithm for OWL-S based semantic search in UDDI," in *Lecture Notes in Computer Science*, vol. 3387, Jan. 2005.

[63] R. Akkiraju, R. Goodwin, P. Doshi, and S. Roeder, "A method for semantically enhancing the service discovery capabilities of UDDI," in *Workshop on Information Integration on the Web IIWeb*, 2003, pp. 87–92.

[64] A. A. Patil, S. A. Oundhakar, A. P. Sheth, and K. Verma, "METEOR-S web service annotation framework," in *WWW '04: Proceedings of the 13th international conference on World Wide Web*. New York, NY, USA: ACM Press, 2004, pp. 553–562.

[65] L. Ding, T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. Doshi, and J. Sachs, "Swoogle: a search and metadata engine for the semantic web," in *CIKM*

'04: Proceedings of the thirteenth ACM international conference on Information and knowledge management. New York, NY, USA: ACM Press, 2004, pp. 652–659.

[66] S. Balzer, T. Liebig, and M. Wagner, "Pitfalls of OWL-S – A practical Semantic Web Use Case," in *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC004)*, New York, NY, USA, Nov. 2004.

[67] T. Payne, M. Paolucci, and D. Martin, "OWL-S 1.1 profile specification," 11 2004. [Online]. Available: http://www.daml.org/services/owl-s/1.1/Profile.owl

[68] P. Mika, D. Oberle, A. Gangemi, and M. Sabou, "Foundations for service ontologies: aligning OWL-S to DOLCE," in *WWW '04: Proceedings of the 13th international conference on World Wide Web.* ACM Press, 2004, pp. 563–572.

[69] C. Masolo, S. Borgo, A. Gangemi, N. Guarino, A. Oltramari, and L. Schneider, "DOLCE : a descriptive ontology for linguistic and cognitive engineering." [Online]. Available: http://www.loa-cnr.it/DOLCE.html

[70] A. Doan, J. Madhavan, P. Domingos, and A. Halevy, "Learning to map between ontologies on the semantic web," in *Proceedings of the Eleventh International WWW Conference*, 2002.

[71] M. Arumugam, A. Sheth, and I. B. Arpinar, "Towards peer-to-peer semantic web: A distributed environment for sharing semantic knowledge on the web," in *Proceedings of the International World Wide Web Conference 2002 (WWW20002)*, Honolulu, Hawaii, USA, 2002.

[72] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, B. L. R. Peter F. Patel-Schneider, and L. A. Stein, "OWL Web Ontology Language Reference," World Wide Web Consortium, Feb. 2004. [Online]. Available: http://www.w3.org/TR/owl-ref/

[73] J. Clark and S. DeRose, *XPath*, World Wide Web Consortium, 1999. [Online]. Available: http://www.w3.org/TR/xpath

[74] V. Lindesay, "Schemaweb directory." [Online]. Available: http://www.schemaweb.info/default.aspx

[75] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz, "An Architecture for a Secure Service Discovery Service," in *Mobile Computing and Networking*, 1999, pp. 24–35.

[76] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area Web cache sharing protocol," University of Wisconsin-Madison, Tech. Rep. 1361, July 1998. [Online]. Available: http://www.cs.wisc.edu/~cao/papers/summary-cache/

[77] E. Sirin, “OWL-S API,” Maryland Information and Network Dynamics Lab Semantic Web Agents Project, Tech. Rep. [Online]. Available: http://www.mindswap.org/2004/owl-s/api/

[78] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, “Jena: implementing the semantic web recommendations,” in *WWW Alt. ’04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters.* ACM Press, 2004, pp. 74–83.

[79] B. Parsia, E. Sirin, M. Grove, and R. Alford, “Pellet OWL reasoner,” Maryland Information and Network Dynamics Lab Semantic Web Agents Project. [Online]. Available: http://www.mindswap.org/2004/owl-s/api/